FUJITSU

FUJITSU Software NetCOBOL V11.0

User's Guide

Linux(64)

J2UL-1950-01ENZ0(00)
October 2014

Preface

NetCOBOL allows you to create, execute, and debug COBOL programs. This manual describes the functions and operations of the NetCOBOL product. The specifics of COBOL syntax are not covered by this manual. Refer to the "NetCOBOL Language Reference" for details of COBOL syntax. It also describes, in detail, the functions of object-oriented programming using COBOL.

Product Names

Names of products described in this manual are abbreviated as:

Product Name	Abbreviation
Oracle Solaris	Solaris
Red Hat(R) Enterprise Linux(R) 6 (for Intel64) Red Hat(R) Enterprise Linux(R) 7 (for Intel64)	Linux(x64)
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 Pro Windows® 8.1 Enterprise	Windows 8.1 or Windows 8.1 (x64)
Windows® 8 Windows® 8 Pro Windows® 8 Enterprise	Windows 8 or Windows 8 (x64)
Windows® 7 Home Premium Windows® 7 Professional Windows® 7 Enterprise Windows® 7 Ultimate	Windows 7 or Windows 7 (x64)

Audience

This manual is for users who develop COBOL programs using NetCOBOL. It assumes users possess the following:

- Basic knowledge of COBOL syntax and
- Basic knowledge of the Solaris or Linux

How this Manual is Organized

This manual consists of the following chapters and appendixes:

Chapter 20 Remote Development Support Function

Explains the remote distributed development support function.

Chapter 21 Operation of CSV (Comma Separated Value) data

How to operate CSV data.

Appendix A Compiler Options

Details of the COBOL 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 COBOL compiler.

Appendix D Intrinsic Function List

Lists all the COBOL intrinsic functions supported by COBOL.

Appendix E Environment Variable List

Lists all the environment variables in this product.

Appendix F Writing Special Literals

How to write special literals for native system.

Appendix G Subroutines Offered by NetCOBOL

Descriptions of subroutines provided by NetCOBOL.

Appendix H Incompatible Syntax Between Standard and Object-Oriented

Descriptions of the combination of the object-oriented and traditional functions.

Appendix I Database Link

Notes on using this product collaboratively with the database.

Appendix J National Language Code

How to create a COBOL applications operating in the national language.

Appendix K ld Command

Descriptions of the ld command syntax and how to use it when combining re-locatable programs generated by the COBOL compiler.

Appendix L Using the make Command

How to use the make command during COBOL development.

Appendix N Security

Guidelines for creating secure applications.

Appendix M COBOL coding techniques

Describes COBOL program coding techniques.

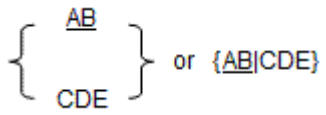
Sample Programs

Examples of programs written in NetCOBOL, and explains how to compile, link, and execute them.

Conventions Used in this Manual

This manual uses the following typographic conventions:

Example of Convention	Description
[See]	Indicates the destination to be seen.
->	Indicates the result of an operation.

Example of Convention	Description
$	Indicates a Bash prompt.
%	Indicates a C shell prompt.
...	Ellipses indicate the item immediately preceding can be specified repeatedly.
ABCDE	Indicates a variable string in a sample program. Replace the variable string with another string. For example: PROGRAM-ID. program name. -> PROGRAM-ID. SAMPLE1.
	Indicates that one of the enclosed items (delimited by |) can be selected. If items are omitted the underlined item is assumed to have been selected.
[ABCDE]	Indicates that the enclosed item may be omitted.

 Note

- In this manual, "Interstage Application Server" is referred to as "Interstage".
- In this manual, OS IV systems including OS IV/MSP and OS IV/XSP are generically called as "OS IV system".

Trademarks

- Linux is a registered trademark of Linus Torvalds.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- X Window System is a trademark of The Open Group.
- 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.
- Microsoft, Windows and Windows Server are registered trademarks of Microsoft Corporation in the U.S.A and/or other countries.
- Other brand and product names are trademarks or registered trademarks of their respective owners.

Acknowledgement

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

The object-oriented language specification for COBOL is currently being discussed as one of the new functions for the Forth COBOL International Standards by ISO/IEC JTC1/SC22/WG4 and NCITS J4 Technology Committees. The object-oriented language specification for COBOL is based on the draft standards resulting from the efforts made to date by the committee. We would like to express our special thanks to those committees for their efforts and dedication, now that we offer COBOL.

The language specifications related to object-oriented programming are subject to change since upgrading efforts on the standard is still in progress. Fujitsu will continue to support updates of these standards until they are finalized, in addition to those described in "NetCOBOL Language Reference."

Export Regulation

Exportation/release of this document may require necessary procedures in accordance with the regulations of your resident country and/or US export control laws.

October 2014

Contents

Chapter 1 Overview

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 Functions

This section explains the COBOL functions and the various utilities provided by this product.

1.1.1 COBOL Functions

This product has the following COBOL functions:

- Nucleus
- Sequential file
- Relative file
- Indexed file
- Inter-program communication
- Sort-merge
- Source text manipulation
- Presentation file
- System program description (SD)
- Command line argument handling
- Environment variable operation
- Intrinsic function
- Floating-point handling
- Object oriented COBOL programming

This product provides the following functions that can be used in server-side applications. Server-side applications run in server-centric environments.

- Multithread

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

1.1.2 Intrinsic fun Programs and Utilities Provided by This Product

This product provides the programs and the utilities listed below. Each program or utility is then summarized.

Table 1.1 COBOL programs and utilities

Name	Purpose
COBOL Compiler	Compiles a described program using COBOL.
COBOL Runtime System	Executes a COBOL application.
COBOL FILE UTILITY	Processes COBOL data files.

COBOL Compiler

The COBOL compiler compiles COBOL source programs to create object programs. The compiler provides the following service functions:

- Generates compiler listings.

- Checks standard specifications.
- Provides global optimization.
- Provides linkage with PowerFORM (form descriptors).

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

COBOL Runtime System

When you execute an application program created with COBOL (from now on referred to as COBOL applications), the COBOL runtime system is called.

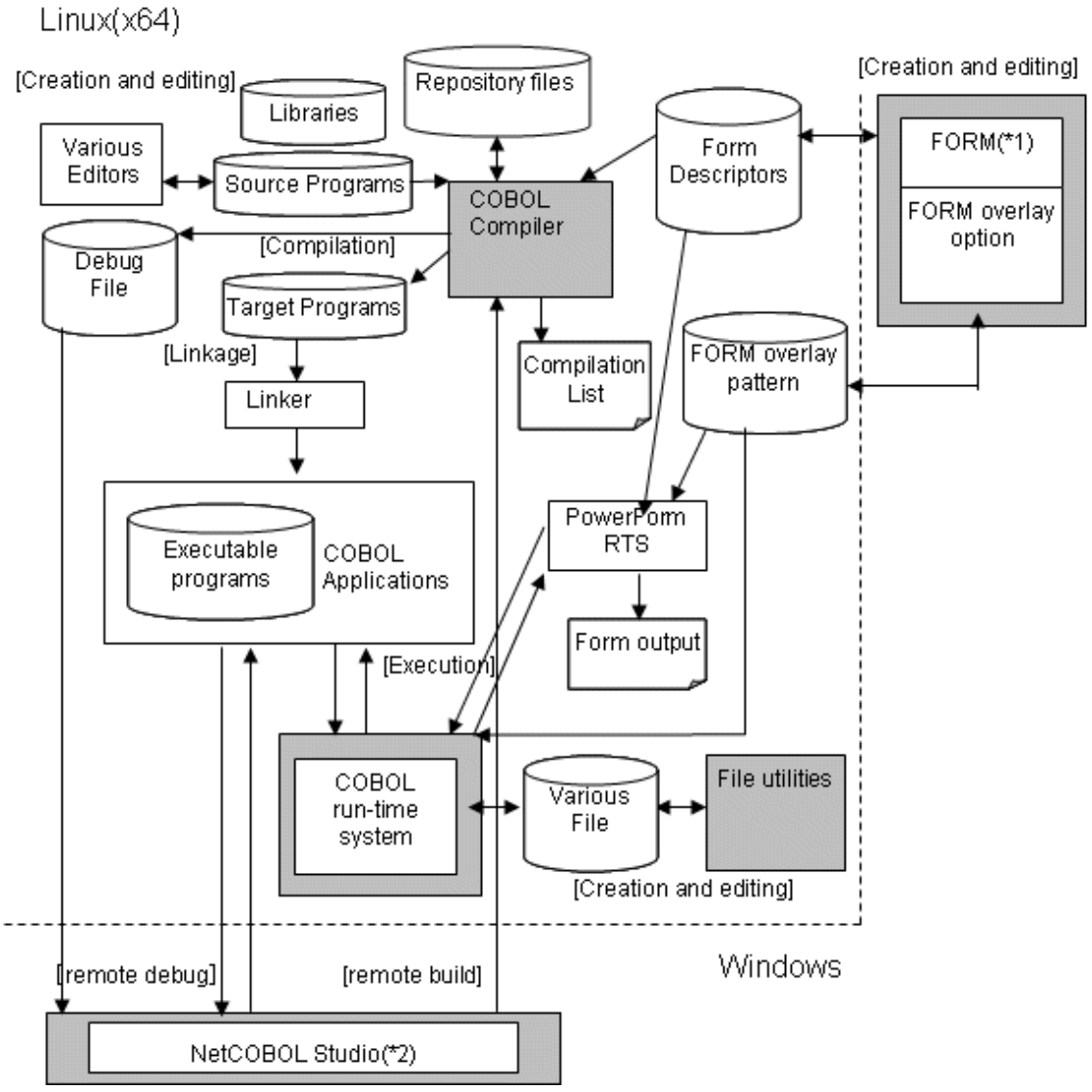
COBOL FILE UTILITY

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

1.2 Development Environment

The following figure provides an overview of the COBOL development environment:

Figure 1.1 COBOL development environment



*1 : The FORM and FORM overlay options are products that operate under 32-bit Windows.

*2 : NetCOBOL Studio is included in the NetCOBOL development products that operate under 32-bit Windows.

1.3 Resource List

The following table shows resource list in this product.

Table 1.2 Resource List

Resource name	Contents	Main Component	File name Rule to be named	Recom. Ext.	Template File name
COBOL source	COBOL source program.	Compiler	Optional	cob cobol	-
COBOL library	COBOL library text.	Compiler	Optional	cbl	-
Form descriptor	Form definition information.	FORM PowerForm RTS	Optional	pmd pxd	-
Form overlay	Form overlay pattern information.	FORM PowerForm RTS	Optional	ovd	-
Makefile	Descriptions of Make-related dependencies and related items	cobmkmf	Optional	Mk	-
Source analysis information file	Source analysis information	Compiler	source-name.sai	sai	-
Compiler option	COBOL compilation option character strings.	Compiler	Optional	cbi	-
Repository	Related class information.	Compiler	Class name.rep	rep	-
Object	Object programs.	Compiler	Source name.o	o	-
Shared object	Shared object programs of subprograms.	Compiler	lib library name.so	so	-
Executable file	Executable program	Compiler	Optional (a.out)	out 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	COBOL.CBR (Optional)	CBR	-
Printing information	Printing format definition such as printer types.	Runtime system	Optional	-	prtinffile
Font table	Font number definition used with printing files.	Runtime system	Optional	-	fonttable

Resource name	Contents	Main Component	File name Rule to be named	Recom. Ext.	Template File name
FCB	Definition of attribution such as number of lines per page, line spacing, and lines to be started.	Runtime system	Optional (4 characters)	-	FCB1
Class information	Specifying 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 (generation ago)	Trace information of a generation ago.	Runtime system	Execution format name.tro	tro	-
COUNT information	Statistics information used with COUNT function.	Runtime system	Optional	-	-
Printer information	Printer information when forms are printed.	PowerForm RTS	Optional	-	PowerForm RTS prc
Debugging information	Debug information for the remote debug function.	Debugger	Source name.svd	svd	-

Chapter 2 Describing a Program

This chapter provides basic information for writing a NetCOBOL program, including editing tips and source program structure. The chapter also describes the difference between fixed and variable formats, explains how to copy library files, and introduces the compiler directing statement.

2.1 Creating and Editing a Program

You can create COBOL source programs (also referred to as source programs) and library text with text editors. This section explains how to create and edit the COBOL source programs and describes library text.

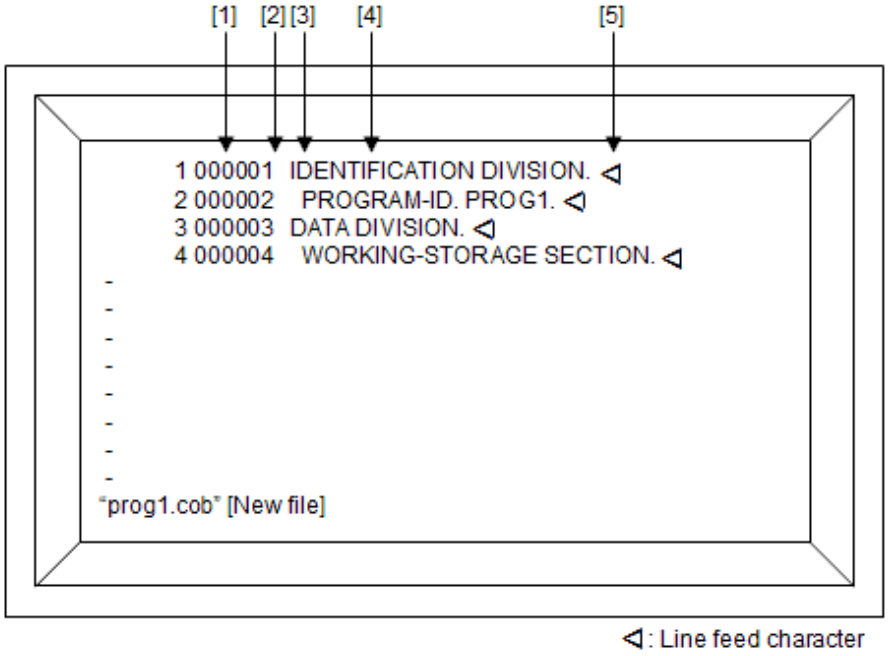
2.1.1 Creating a Source Program

This section explains how to create and edit the source program.

The format of source program lines is stipulated by the COBOL reference format. To create source programs with editors, you must 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:

Figure 2.1 Edit Screen



- [1] Sequence number area (columns 1 to 6)

  Specify line numbers in the sequence number area. Line numbers can be omitted by entering blank spaces.

- [2] Indicator area (column 7)

  Use the indicator area when continuing a line or to change a line to a comment.

- [3] Area A (columns 8 to 11)

  The columns 8 to 11 are called area A.

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

- [4] Area B (column 12 and later)

  Column 12 and later is called area B.

COBOL statements and data items whose level-number is not 77 or 01 are described in this Area B.

- [5] Line feed character (end of line)

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

Information

Tab characters (ASCII X"09") can be specified as the nonnumeric literal in a source program. A tab character occupies 1 byte within the non-numeric literal.

2.1.2 Creating Library Text

You can copy library text to a source program using the COPY statement. The library text is created using utilities or in the same manner as you create source programs, using text editors such as vi.

The reference format (fixed, variable, or free) for the library text does not have to be the same as the format of the source program that copies the library text. However, when a single program copies several library texts, the reference formats for each library text must be identical. Specify the reference format of the library texts using the compiler option in the same manner as you specify source programs.

Normally, the name of the file in which the library text is to be stored is "library-text-name.cbl".

Information

Form descriptor can use any reference format method.

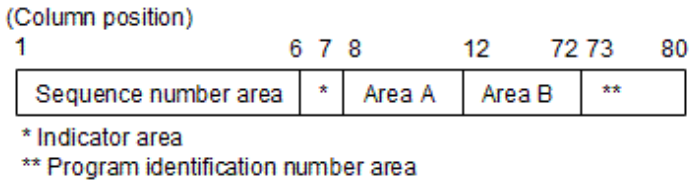
2.2 Program Format

Each line of a COBOL source program is delimited with a line feed character. When creating a COBOL source program, the format of a line has to be described in accordance 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 specifically declare the compiler option SRF. Refer to "A.2.43 SRF (reference format type)".

Each format is described below. Line feed characters are not assumed to be a part of the line.

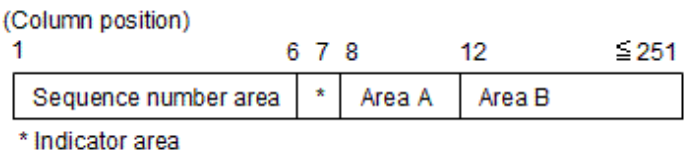
Fixed Format

In the fixed format, each line in the COBOL source program has a fixed length of 80 bytes. The following figure shows the reference format of the fixed format:



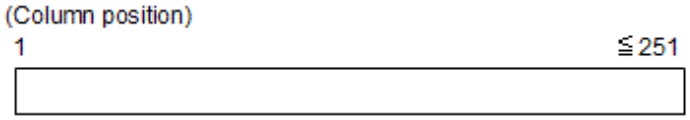
Variable Format

In the variable format, each line in the COBOL source program can be up to 251 bytes long. The following figure shows the reference format of the variable format:



Free Format

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



2.3 Compiler Directing Statements

Compiler options can be declared in the source programs using a compiler directing statement. The compiler directing statement has the following format:

```
@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 (,).
- 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 in the compiler directing statement.

Example

```
000100 @OPTIONS MAIN,APOST
000200 IDENTIFICATION DIVISION.
000300  PROGRAM-ID. PROG1.
      *    :
```

Note

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

Chapter 3 Compiling and Linking Programs

This chapter explains how to compile and link NetCOBOL programs. It includes information on how to use library text, the COPY statement, and the compiling and linking main program and subprogram requirements. This chapter also describes compilation requirements when using the demote debug function of NetCOBOL Studio and compiler resources.

3.1 Compilation and Link

Compile and link a COBOL source program to create an executable program. This section explains the environment variables to be set at compilation, basic compilation and linkage operation of a source program, how to link a subprogram, and resources necessary for the COBOL compiler.

3.1.1 Environment Variables to be Set at Compilation

You can use environment variables to set options or file names to be specified every time you compile the source program. The following environment variables can be set before you compile the source program:

- COBOLOPTS (Specification of compiler options)
- COBCOPY (Specification of library file directory)
- COB_COPYNAME (Specification of search condition of library text)
- COB_LIBSUFFIX (Specification of extension of library file)
- SMED_SUFFIX (Specification of extension of form descriptor file)
- FORMLIB (Specification of form descriptor file directory)
- COB_REPIN (Specification of repository file input destination directory)

3.1.1.1 COBOLOPTS (Specification of compiler options)

Set the cobol command compiler options.

3.1.1.2 COBCOPY (Specification of library file directory)

Specify the directory name where the library files are stored.

Separate multiple directories with colons (:). In this case, the directories are retrieved in the order specified.

See

3.3.1.11 -I (Specification of library file directory)

3.1.1.3 COB_COPYNAME(Specification of search condition of library text)

Specify how to search for the name of the file that contains the library text and form descriptor. Specify one of the following:

- Upper

  Searches for all upper-case file names.

- Lower

  Searches for all lower-case file names.

- Default

  Searches for file name of "[library text name].[lower-case extension]" format. The management of lower-case and upper-case letters in the library text name depends on the compile option ALPHAL specification. When no environment variables are set, the Default is assumed to have been set.

When the file name in the COPY statement is specified by the constant string, the environment variable COB_COPYNAME specification is ignored, and is searched by the same constant string as the specification.

3.1.1.4 COB_LIBSUFFIX (Specification of extension of library file)

Specify an optional character string to be set the extension for library files.

Separate multiple extensions with commas (,). In this case, the files are retrieved in the order specified.

3.1.1.5 SMED_SUFFIX (Specification of extension of form descriptor file)

Specify an optional character string to be the extension for form descriptor files.

Separate multiple extensions with commas (,). In this case, the files are retrieved in the order specified.

3.1.1.6 FORMLIB (Specification of form descriptor file directory)

Specify the directory name where the form descriptor files are stored.

Separate multiple directories with colons (:). In this case, the directories are retrieved in the order specified.

3.3.1.14 -m (Specification of form descriptor file directory)

3.1.1.7 COB_REPIN (Specification of repository file input destination directory)

Specify the directory name where the input repository files are stored.

Separate multiple directories with colons (:). In this case, the directories are retrieved in the order specified.

3.3.1.16 -R (Specification of repository file input destination directory)

3.1.2 Basic Operation

There are four ways to compile and link the source program to create an executable program:

- Use the cobol command to compile and link the program all at once.
- Use the cobol command to compile the program, then link the program with the cobol command.
- Use the cobol command to compile the program, then link the program with the ld command.

When you use the cobol and ld command to compile and link the program, the make command can be used to create the executable program more efficiently. For the details on the make command, refer to "Appendix L Using the make Command".

The following explains how to compile and link the program with the cobol and ld commands.

In addition, when using the cobol command, you need to have the directory that stores the COBOL compiler set in the following environment variables:

- PATH
- LD_LIBRARY_PATH

How to Compile and Link with the cobol Command All At Once

The cobol command compiles the source program to create re-locatable programs, and then links the re-locatable programs to create an executable program. Thus, you can create an executable program with only the cobol command, without using the ld command. This is useful because users do not have to watch for various library subroutines the product provides.

For the details on the parameter syntax of the cobol command, refer to "3.3 cobol Command".

The following shows an example of compilation and linkage with the cobol command:

Example

```
$  cobol -dy -M -o P1 P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

- Input

  P1.cob (source file)

- Output

  P1.o (object file)

  P1 (executable file)

- Option

  -dy (specifying the dynamic linkage)

  -M (specifying the main program)

  -o (output destination of the executable program)

How to Compile, then Link with the cobol Command

The cobol command compiles the source program to create a re-locatable program without producing an executable program. It also links the re-locatable program to the shared object program to create an executable program. When you link the programs using the cobol command, you do not have to specify the library subroutines the product provides respectively as with compilation and linkage with the cobol command.

The following is an example of compilation with the cobol command and linkage:

Example

```
$  cobol -c -M P1.cob      <-  (Compilation)
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$  cobol -dy -o P1 P1.o    <-  (Linkage)
```

cobol command (Compilation)

- Input

  P1.cob (source file)

- Output

  P1.o (object file)

- Option

  -M (specifying the main program)

  -c (specifying to compile only)

cobol command (Linkage)

- Input

  P1.o (object file)

- Output

  P1 (executable file)

- Option

  -dy (specifying the dynamic linkage)

  -o (output destination of the executable program)

How to Compile with the cobol Command and Link with the ld Command

You can create a re-locatable program with the cobol command, then create an executable program with the ld command.

For the details on how to execute the ld command, refer to "Appendix K ld Command".

Note

Compiler errors

- The object program will be created even if I, W, or E-level compile errors occur. Check the content of the error message when the compilation has finished.
- When the cobol command in a Makefile outputs an E-level or above compile error during executing make command, the return code indicates that make command cannot continue any further and will be terminated. Note that the object program may not be generated in this case.

3.1.3 How to Compile the Program Using the Library (COPY Statement)

This section explains how to compile the source program with the COPY statement.

When you compile the source program with the COPY statement described, you need a file that contains the library text to be derived with the COPY statement (referred to as a Library file hereafter). The name of the library file is determined by the text-name or text-name literal specified in the COPY statement.

When you describe the text-name in the COPY statement or describe the text-name literal as a relative path name, the directory containing the library file must be provided to the COBOL compiler.

The following shows how to specify the directory where the library file is stored in the order of descending priority:

When Describing the COPY Statement Without IN/OF Specified in the Source Program

1. Specify the -I option that specifies the library file directory as an operand of the cobol command.
2. Set the library file directory in the environment variable COBCOPY. For the details of COBCOPY, refer to "3.1.1.2 COBCOPY (Specification of library file directory)".

When Describing the COPY Statement With IN/OF Specified in the Source Program

Set the library file directory in the environment variable with the same name as that of the library-name specified in IN/OF phrase. If this is not set, a compile error occurs at compilation of the program.

Using the environment variable COB_COPYNAME enables upper or lower-case characters, and the default specification when searching the library file and form descriptor file. When no environment variables are set, the default settings apply. For the details of COB_COPYNAME, refer to "3.1.1.3 COB_COPYNAME(Specification of search condition of library text)".

The following are examples of running the cobol command when compiling the program with the library:

Example

When the Source File and Library File are in the Same Directory

[When using COPY A. in the source program]

```
$  cobol -dy -M -o P1 P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

- Input

  P1.cob (source file)

  A.cbl (library file)

- Output

  P1.o (object file)

  P1 (executable file)

- Option

  -dy (specifying the dynamic linkage)

  -M (specifying the main program)

  -o (output destination of the executable program)

When the Source File and Library File are in Separate Directories

[When using COPY A. in the source program]

```
$  cobol -dy -M -o P1 -I/home/COBOL P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

or

```
$  COBCOPY=/home/COBOL ; export COBCOPY
$  cobol -dy -M -o P1 P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

- Input

  P1.cob (source file)

  /home/COBOL/A.cbl (library file)

- Output

  P1.o (object file)

  P1 (executable file)

- Option

  -dy (specifying the dynamic linkage)

  -M (specifying the main program)

  -o (output destination of the executable program)

  -I (input destination of the library file)

When Using COPY A OF B. in the Source Program

```
$  B=/home/COBOL ; export B
$  cobol -M -o P1 P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

- Input

  P1.cob (source file)

  /home/COBOL/A.cbl (library file)

- Output

  P1.o (object file)

P1 (executable file)

- Option

  -M (specifying the main program)

  -o (output destination of the executable program)

Note

If you use lower-case alphanumeric characters in the name of library text file in the source program, the case of the file derived depends on the specification of compiler option ALPHAL/NOALPHAL.

For example:

If the library text file COPY a. is used in the source program:

- If the compiler option ALPHAL is specified, the file derived will be A.cbl.
- If the compiler option NOALPHAL is specified, the file derived will be a.cbl.

3.1.4 How to Compile and Link the Program that Calls a Subprogram

This section explains how to compile and link a program that calls a subprogram and the subprogram that is called to create an executable program. Note that programs that call the subprogram are referred to as Main Programs and programs that are called are referred to as subprograms.

For the details on how to link the programs, including information on linkage types, refer to "3.2.2 Linkage Type and Program Structure".

Before you compile and link the main program to create an executable program, compile and link the subprogram to create a shared object program. The shared object program is required when you link the main program.

The following is an example of creating an executable program in static linkage and dynamic linkage:

Example

If you compile and link the three source programs: P1, P2, and P3, the relationship between the programs is as follows:

- P1 calls P2 and P3.
- P2 and P3 call no programs.

Static Linkage

```
$  cobol -c P2.cob P3.cob                              [1]
HIGHEST SEVERITY CODE = I
HIGHEST SEVERITY CODE = I
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=2
$  cobol -dn -M -o P1 P1.cob P2.o P3.o                 [2]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

[1] Compile the subprogram to create a re-locatable program.

- Input

  P2.cob (source file)

  P3.cob (source file)

- Output

  P2.o (object file)

  P3.o (object file)

- Option

  -c (specifying to compile only)

[2] Compile and link the main program to create an executable program.

- Input

  P1.cob (source file)

  P2.o (object file)

  P3.o (object file)

- Output

  P1 (executable file)

- Option

  -dn (specifying the static linkage)

  -M (specifying the main program)

  -o (output destination of the executable program)

Dynamic Linkage

```
$  cobol -dy -shared -o libP2.so P2.cob               [1]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$  cobol -dy -shared -o libP3.so P3.cob               [2]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$  cobol -dy -M -o P1 -L. -lP2 -lP3 P1.cob            [3]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

[1] [2] Compile and link the subprogram to create a shared object file.

- Input

  P2.cob (source file)

  P3.cob (source file)

- Output

  P2.o (object file)

  P3.o (object file)

  libP2.so (shared object file)

  libP3.so (shared object file)

- Option

  -dy (specifying the dynamic linkage)

  -G or -shared (specifying to create a shared object program)

  -o (output destination of the shared object program)

[3] Compile and link the main program to create an executable file.

- Input

  P1.cob (source file)

  libP2.so (shared object file)

  libP3.so (shared object file)

- Output

  P1 (executable file)

- Option

  -dy (specifying the dynamic linkage)

  -M (specifying the main program)

  -o (output destination of the executable program)

  -L (Library search path name)

  -l (Library to be linked)

3.1.5 How to Compile the Program When Using the remote debug function of NetCOBOL Studio

Specify the -Dt option at compilation and linkage when you use the remote debug function of NetCOBOL Studio. Compiling a program with the -Dt option specified will create a debugging information file, which is required to debug a program with the remote debug function of NetCOBOL Studio. Generally the debugging information file is stored in the directory where the source file is stored. You can specify the directory to which the debugging information file is to be stored with the -dd option. The source file name is separated with a period (.) with the character string svd followed in the name of the debugging information file.

Example

```
$  cobol -M -WC,"NOOPTIMIZE" -o P1 -Dt P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

- Input

  P1.cob (source file)

- Output

  P1.o (object file)

  P1 (executable file)

  P1.svd (debugging information file)

- Option

  -M (specifying the main program)

  -o (output destination of the executable program)

  -Dt (specifying to use the remote debug function of NetCOBOL Studio)

3.1.6 Files Used by the NetCOBOL Compiler

The COBOL compiler uses the following files:

- Source file (*.cob)
- Object file (*.o)
- Library file (*.cbl)
- Form descriptor file (*.smd/*.pmd)
- Compile list file (optional/*.lst)
- Repository file (*.rep)
- Source analysis information file (*.sai)
- Shared object file (lib*.so)
- Executable file (optional/a.out)

- Debugging information file (*.svd)
- Option file (optional)

The recommended extensions are outlined in Table below:

Table 3.1 Files used in the cobol Command

File	File Contents	File Name Format	I/O	Condition to Use	Related Compiler Option Related Environment Variable
Source file	Source program	program name.cob (*1)	I	Required.	- -
Object file	Target program (re-locatable program)	source file name.o (*2)	I	When linking the program.	- -
			O	Created when compilation has completed successfully.	-do -
Library file	Library text	library text name.cbl (*3)	I	When compiling a source program that uses the library text.	-I COBCOPY COB_LIBSUFFIX library text name COB_COPYNAME
Form descriptor file	Form descriptor	Form descriptor name.smd Form descriptor name.pmd (*4)	I	When compiling a source program that uses a form descriptor.	-m SMED_SUFFIX FORMLIB
Compile list file	Compile list	Optional source file name.lst (*5)	O	Outputs the compile list to the file.	-P -
Repository file	Class-related information for inheritance and consistence check	class name.rep	I	When compiling a source program that has a REPOSITORY paragraph.	-R COB_REPIN
			O	When compiling a source program with classes defined.	-dr -
Source analysis information file	Information for source analysis using SIMPLIA or such software	source file name.sai	O	When analyzing the source.	-ds -
Shared object file	Shared object program of a subprogram	libsub program name.so (*6)	I	When linking the program.	-l -
			O	Created when compilation has completed with -shared option specified.	-shared -o -

File	File Contents	File Name Format	I/O	Condition to Use	Related Compiler Option Related Environment Variable
Executable file	Executable program	Optional The default is a.out	O	Created when linkage has completed with -shared option not specified.	-o -
Debugging information file	Debugging information for the remote debug function of NetCOBOL Studio	source file name.svd (*7)	O	Created when compilation has completed with -Dt option specified.	-Dt -dd TEST -
Option file	Character string that indicates compiler options	Optional	I	When specifying compiler options stored in a file.	-i -

- *1 : Upper-case COB, CBL, and COBOL are not treated as extensions. You can specify any file name, however the product outputs files with the following extensions, which cannot be used for the COBOL source file names:
  - lst
  - rep
  - sai
- *2 : If the name of a COBOL source file has any of the following extensions, the extension will be changed to 'o' in the file name: cob, cbl or cobol. Otherwise, the extension 'o' will be attached to the COBOL source file name.
- *3 : The extension 'cbl' can be changed to an optional character string with the environment variable COB_LIBSUFFIX. If character string 'None' is set, no extensions are assumed to have been specified. If environment variable COB_LIBSUFFIX is not specified, library files are searched in the order of extension (1) cbl, (2) cob, and (3) cobol.

  See "3.1.1.4 COB_LIBSUFFIX (Specification of extension of library file)".

```
$  COB_LIBSUFFIX = character string ; export COB_LIBSUFFIX
```

- *4 : The file extension 'smd' can be changed to an optional character string with the environment variable SMED_SUFFIX. If character string 'None' is set, no extensions are assumed to have been specified. If environment variable SMED_SUFFIX is not specified, form definition files are searched in the order of extension (1) pmd and (2) smd.

  See "3.1.1.5 SMED_SUFFIX (Specification of extension of form descriptor file)".

```
$  SMED_SUFFIX = character string ; export SMED_SUFFIX
```

- *5 : If you specify omitting the file name (-) with -P option, the format of the compile list file name is as follows:
  - If the name of a COBOL source file uses any of the following extensions, the extension will be changed to 'lst' in the file name: cob, cbl or cobol.
  - Otherwise, the extension 'lst' will be attached to the COBOL source file name.
- *6 : Users need to specify this explicitly at linkage of the program.
- *7 : If the name of a COBOL source file has any of the following extensions, the extension will be changed to 'svd' in the file name: of cob, cbl or cobol. Otherwise, the extension svd will be attached to the COBOL source file name.

3.1.7 Information Provided by the NetCOBOL Compiler

The NetCOBOL compiler outputs the following information:

- Diagnostic message

- Option information
- Compiler unit statistical information
- Cross-reference
- Source program listing
- Target program listing
- Data area listing

3.1.7.1 Diagnostic Message Listing

The NetCOBOL compiler reports program compilation results as a diagnostic message. The diagnostic message is generally output to where the standard errors are output. You can specify the directory to which the error message is output with the -P option.

```
$  cobol -M -o P1 -PP1.list P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

- Input

  P1.cob (source file)

- Output

  P1.o (object file)

  P1 (executable file)

  P1.list (compile list file)

Diagnostic Message Listing

```
** DIAGNOSTIC MESSAGE ** (SAMPLE1)

JMN25031-S  63 USER WORD 'A' IS UNDEFINED.
```

3.1.7.2 Option Information Listing and Compile Unit Statistical Information Listing

If you want to know what compiler options are in effect during execution of the cobol command, specify the compiler option MESSAGE with the -WC option. Specifying the compiler option MESSAGE will output the option information listing and the compile unit statistical information listing. These lists are generally output to where the standard errors are output. You can specify the directory to which the error message is to be output with -P option.

```
$  cobol -M -o sample1 -Psample1.list -WC,"MESSAGE,LINESIZE(80)" sample1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

- Input

  sample1.cob (source file)

- Output

  sample1.o (object file)

  sample1 (executable file)

  sample1.list (compile list file)

The following explains the option information listing, diagnostic message listing, and the compile unit statistical information listing:

Option Information Listing

```
** OPTIONS SPECIFIED **
MAIN,MESSAGE,LINESIZE(80)
** OPTIONS ESTABLISHED **
  ALPHAL(ALL)             LANGLVL(85)           NOSAI
  ASCOMP5(NONE)           LINECOUNT(0)            SDS
  BINARY(WORD,MLBON)      LINESIZE(80)          NOSHREXT
NOCHECK                 NOLIST                  SMSIZE(0)
NOCONF                    MAIN                  NOSOURCE
NOCOPY                  NOMAP                   SRF(VAR,VAR)
NOCOUNT                   MESSAGE               SSIN(SYSIN)
  CREATE(OBJ)             MODE(STD)             SSOUT(SYSOUT)
  CURRENCY($)           NONAME                  STD1(JIS2)
NODBC                     NCW(STD)              TAB(8)
NODLOAD                   NSPCOMP(NSP)          NOTEST
  DUPCHAR(STD)          NONUMBER                  THREAD(SINGLE)
NOEQUALS                  OBJECT                NOTRACE
  FLAG(I)                 OPTIMIZE              NOTRUNC
NOFLAGSW                  QUOTE                 NOXREF
NOINITVALUE               RCS(UTF16,LE)           ZWB
NOLALIGN                  RSV(ALL)
```

Compile Unit Statistical Information Listing

```
** STATISTICS **

  FILE NAME     = sample1.cob
  SOURCE NAME   = SAMPLE1
  DATE AND TIME = WED JAN 6 18:44:06 2010 (GMT-7.00)

    SOURCE STATEMENTS              =         66 RECORDS
    PROGRAM SIZE(.TEXT)            =         1688 BYTES
    PROGRAM SIZE(.DATA)            =          456 BYTES
    CONTROL LEVEL                  =         0101 LEVEL
    CPU TIME                       =       0.18 SECONDS

  HIGHEST SEVERITY CODE = I
```

3.1.7.3 Cross Reference List

Specifying compiler option XREF will output the cross reference list.

Cross reference list output format

```
     DEFINITION  NAME                        ATTRIBUTES AND REFERENCE
       [1]  [2]                              [3]
        39  COUNTER                          54S 55R 56R 63R
        61  DISPLAY-SECTION
        40  INPUT-CHARACTER                  49S 56R
        41  INPUT-REQUEST-MESSAGE            47R
        45  INPUT-SECTION
         3  SAMPLE1
        51  SEARCH-SECTION
        37  TOP-CHARACTER                    56R
        36  WORD                             63R
         7  WORDLIST                         35D
        35  WORDTABLE
```

Explanation of the cross reference list

- [1] DEFINITION

  Indicates line number in the following format:

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

  - COPY qualification-value

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

  - Line number

    Line numbers where a name is defined as an asterisk (*) indicate the name was implicitly defined.

  - Mark for name defined in separate compilation

    Line numbers with a hash mark (#) indicate the name is defined in a separate compilation.

- [2] NAME

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

- [3] ATTRIBUTES AND REFERENCE

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

  - A: A CALL statement, INVOKE statement, and In-line invocation parameter.
  - D: Reference by identification division, environment division, and data division.
  - P: Reference by PERFORM statement.
  - R: Reference by procedure division.
  - S: Setting.

See "A.2.52 XREF (whether a cross-reference list should be output)".

3.1.7.4 Source Program Listing

To output a source program listing to the compilation list file, specify the SOURCE compile option.

Source program listing output format

```
   Line number  Sequence number A   B
      [1]              [2]
       1     000100 IDENTIFICATION DIVISION.
       2     000200  PROGRAM-ID. A.
       3     000300*
       4     000400 DATA DIVISION.
       5     000500 WORKING-STORAGE SECTION.
       6     000600     COPY  A1.
     1-1 C   000600 77 answer      PIC 9(2).
     1-2 C   000700 77 divisor     PIC 9(2).
     1-3 C   000800 77 dividend    PIC 9(2).
       7     000900*
       8     001000 PROCEDURE DIVISION.
       9     001100*
      10     001200     MOVE 10 TO dividend.
      11     001300     MOVE  0 TO divisor.
      12     001400*
      13     001500     COMPUTE answer = dividend / divisor.
      14     001600*
      15     001700       EXIT PROGRAM.
      16     001800   END PROGRAM A.
```

Explanation of the source program listing

- [1] Line number

  Line numbers are displayed in the following format:

  - If the NUMBER compile option is enabled:

    ```
    [COPY-qualification-value-] user-line-number
    ```

    - COPY-qualification-value

      This identification number is added to lines with library text identification numbers embedded in the source program or sequence numbers not sorted in ascending order. Starting from 1 and incremented in steps of one, it is assigned to a COPY statement or sequence number not sorted in ascending order.

    - user-line-number

      Values in the sequence number area of the source program are used. If the sequence number area includes non-numeric characters, the sequence number of each line is changed to the correct sequence number immediately before the number is increased by 1. In addition, even if identical sequence numbers are successively assigned, they are not regarded as an error and are used as is.

  - If the NONUMBER compile option is enabled:

    ```
    [COPY-qualification-value-] intra-source-file-relative-number
    ```

    - COPY-qualification-value

      This identification number is added to library text identification numbers embedded in the source program. Starting from 1 and incremented in steps of one, it is assigned to a COPY statement.

    - intra-source-file-relative-number

      The compiler assigns it as line numbers in ascending order starting from 1 and incremented in steps of one. The value assigned to the source embedded by a COPY statement is a value starting from 1 and incremented in steps of one, and indications that it has been embedded by the COPY statement ("C" indications) are placed in line numbers and the source program.

- [2] The source program is displayed.

3.1.7.5 Target Program Listing

To output a target program listing into the compilation list file, specify the LIST compile option.

Target program listing when OPTIMIZE is enabled (default)

```
  ADDRESS     OBJECT CODE   LABEL       INSTRUCTION
                            GLB.10 [4]
-- 12 ---[5]         MOVE [6]
[1]          [2]                         [3]
000000000453  4C8BABB0010000   mov       r13,qword ptr [rbx+0x000001B0]    BCO.0
00000000045A  498B4517         mov       rax,qword ptr [r13+0x17]          +000000000
00000000045E  488983F0010000   mov       qword ptr [rbx+0x000001F0],rax    TOTAL
000000000465  498B4D18         mov       rcx,qword ptr [r13+0x18]          +000000000+1
000000000469  48898BF1010000   mov       qword ptr [rbx+0x000001F1],rcx    TOTAL+1
000000000470  410FB6451F       movzx     eax,byte ptr [r13+0x1F]           +000000000+8
000000000475  80A3F80100000F   and       byte ptr [rbx+0x000001F8],0x0F    TOTAL+8
00000000047C  24F0             and       al,-0x10
00000000047E  0883F8010000     or        byte ptr [rbx+0x000001F8],al      TOTAL+8
--- 13 ---            MOVE
000000000484  49C7C600000000   mov       r14,0x00000000
--- 14 ---            PERFORM
00000000048B  4C8BBB98010000   mov       r15,qword ptr [rbx+0x00000198]    BVA.1
000000000492  458B27           mov       r12d,dword ptr [r15]              DAYS
000000000495  410FCC           bswap     r12d
000000000498  4D63E4           movsxd    r12,r12d
00000000049B  4D8BDC           mov       r11,r12
00000000049E  4C899C24C0010000 mov       qword ptr [rsp+0x000001C0],r11    PCT.1
0000000004A6  4489B3FC010000   mov       dword ptr [rbx+0x000001FC],r14d   IDX
```

```
                                BBK=00005(001)
                                .........LN.LX
                           GLB.11 [6]
0000000004AD  4C63BBFC010000   movsxd    r15,dword ptr [rbx+0x000001FC]    IDX
0000000004B4  4C8BAC24C0010000 mov       r13,qword ptr [rsp+0x000001C0]    PCT.1
0000000004BC  4983FD00         cmp       r13,0x00
0000000004C0  0F8E03020000     jle       0x0000000006C9                    GLB.17
                                BBK=00006(001)
0000000004C6  4C89E9           mov       rcx,r13
0000000004C9  4883E901         sub       rcx,0x01
0000000004CD  4989CC           mov       r12,rcx
0000000004D0  4C89A424C0010000 mov       qword ptr [rsp+0x000001C0],r12    PCT.1
```

Target program listing when NOOPTIMIZE is enabled

```
  ADDRESS     OBJECT CODE   LABEL       INSTRUCTION
--- 12 ---            MOVE
000000000452  4C8BBBB0010000   mov       r15,qword ptr [rbx+0x000001B0]    BCO.0
000000000459  498B4717         mov       rax,qword ptr [r15+0x17]          +000000000
00000000045D  488983F0010000   mov       qword ptr [rbx+0x000001F0],rax    TOTAL
000000000464  498B4F18         mov       rcx,qword ptr [r15+0x18]          +000000000+1
000000000468  48898BF1010000   mov       qword ptr [rbx+0x000001F1],rcx    TOTAL+1
00000000046F  410FB6471F       movzx     eax,byte ptr [r15+0x1F]           +000000000+8
000000000474  80A3F80100000F   and       byte ptr [rbx+0x000001F8],0x0F    TOTAL+8
00000000047B  24F0             and       al,-0x10
00000000047D  0883F8010000     or        byte ptr [rbx+0x000001F8],al      TOTAL+8
--- 13 ---            MOVE
000000000483  49C7C600000000   mov       r14,0x00000000
00000000048A  4489B3FC010000   mov       dword ptr [rbx+0x000001FC],r14d   IDX
--- 14 ---            PERFORM
000000000491  4C8BA398010000   mov       r12,qword ptr [rbx+0x00000198]    BVA.1
000000000498  458B2C24         mov       r13d,dword ptr [r12]              DAYS
00000000049C  410FCD           bswap     r13d
00000000049F  4D63ED           movsxd    r13,r13d
0000000004A2  4D8BDD           mov       r11,r13
0000000004A5  4C899C24B0010000 mov       qword ptr [rsp+0x000001B0],r11    PCT.1
                           GLB.11
0000000004AD  4C8BBC24B0010000 mov       r15,qword ptr [rsp+0x000001B0]    PCT.1
0000000004B5  4983FF00         cmp       r15,0x00
0000000004B9  0F8E03020000     jle       0x0000000006C2                    GLB.17
0000000004BF  4C89F9           mov       rcx,r15
0000000004C2  4883E901         sub       rcx,0x01
0000000004C6  4989CE           mov       r14,rcx
0000000004C9  4C89B424B0010000 mov       qword ptr [rsp+0x000001B0],r14    PCT.1
```

- [1] Address

  Indicates the relative offset from the top of the machine language object.

- [2] Machine language command code

  Displays machine language codes (object codes).

- [3] Assembler format instruction

  Displays machine language statements in a format compatible with the assembly language for x64 architecture.

- [4] Procedure name and procedure number

  Displays procedure names and procedure numbers generated by the compiler.

- [5] Line number

  Displays COBOL program line numbers.

- [6] Verb name

  Displays the names of verbs coded in the COBOL program.

3.1.7.6 Data Area Listings

To output a data area listing into the compilation list file, specify the MAP compile option.

The following types of data area listings are available:

- Data map listing
- Program control information listing
- Session size listing

Data map listing output format

```
  [1]  [2]                  [3]  [4]    [5]        [6]     [7]     [8]    [9]   [12]
 LINE  ADDR    OFFSET REC-OFFSET LVL NAME        LENGTH ATTRIBUTE   BASE   DIM   ENCODING
[10]**MAIN**
 15  heap+00000240               FD OUTFILE                LSAM   BHG.000000
 16 [heap+000001F8]+00000000   0 01 PRINT RECORD     60 ALPHANUM   BVA.000003    UTF8
 18  heap+000003C0             0 77 ANSWER            8 EXT-DEC    BHG.000000
 19  heap+000003C8             0 77 DIVISOR           4 EXT-DEC    BHG.000000
 21  rdat+00000040             0 01 CSTART            8 ALPHANUM   BCO.000000    UTF8
 22  rdat+00000048             0 01 CEND              8 ALPHANUM   BCO.000000    UTF8
 23  rdat+00000050             0 01 CRES              8 ALPHANUM   BCO.000000    UTF8
 25 [heap+000001E8]+00000000   0 01 DIVIDEND          2 EXT-DEC    BVA.000001
[11]**SUB1**
 48 [heap+000001F0]+00000000   0 01 DISPLAY           8 EXT-DEC    BVA.000002
```

Explanation of the data map listing

For data coded in the data block (working-storage section, file section, constant section, linkage section, and report section) of the source program, information on data area assignment and data attributes in the target program is output.

- [1] Line number

  Displays line numbers in the following format:

  - If the NUMBER compile option is enabled:

    ```
    [COPY-qualification-value-] user-line-number
    ```

  - If the NONUMBER compile option is enabled:

    ```
    [COPY-qualification-value-] intra-source-file-relative-number
    ```

  For details on the COPY qualification value, user line number, and intra-source relative number, see the "3.1.7.4 Source Program Listing".

- [2] Address, Offset

  Displays addresses in the following format indicating the areas of data items assigned in the target program:

  ```
  Section name + Relative address
  ```

  - Section name

    One of the following is displayed:

    ```
    rdat (.rodata)
    data (.data)
    stac  (stack)
    heap (heap)
    hea2 (heap)
    ```

  - Relative address

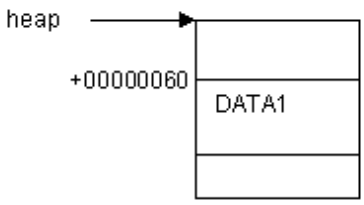
    Provides the relative address from the base address of the section.

    An offset from an address is displayed within "[ ]".

The method of referencing a data area depends on whether an offset is indicated.

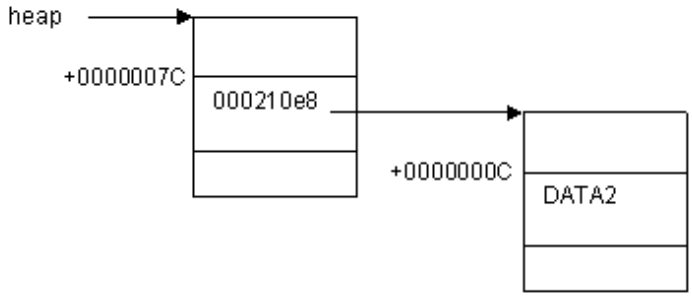
- If no offset is indicated (DATA1: heap+00000060)

  This indicates that the DATA1 data area is located by adding 0x60 to the address stored in heap.



- If an offset is indicated (DATA2: [heap+0000007C]+0000000C)

  An address is stored at the location determined by adding 0x7C to the address stored in heap (in this example, 000210e8). As indicated, the DATA2 data area is located by adding 0x0C to the address. (This example shows that the DATA2 data area is located at 0x000210e8+0x0c.)



- [3] Displacement

  Displays the offset of records in decimal notation.

- [4] Level

  Displays level numbers coded in the source program.

- [5] Name

  Displays the name of data items in the source program. When a name whose length exceeds 30 bytes is displayed, its 31st and subsequent bytes are not displayed.

- [6] Length

  Displays the lengths of data items in decimal notation. File names are not displayed.

- [7] Attribute

  Displays data attributes by using the following symbol.

GROUP-F	Fixed-length group item
GROUP-V	Variable-length group item
ALPHA	Alphabetic
ALPHANUM	Alphanumeric
AN-EDIT	Alphanumeric edited
NUM-EDIT	Numeric edited
INDEX-DATA	Index data
EXT-DEC	Zoned decimal
INT-DEC	Packed-decimal

FLOAT-L	Double-precision internal floating-point
FLOAT-S	Single-precision internal floating-point
EXT-FLOAT	External floating-point
BINARY	Binary
COMP-5	Binary
INDEX-NAME	Index name
INT-BOOLE	Internal Boolean
EXT-BOOLE	External Boolean
NATIONAL	National
NAT-EDIT	National edited
OBJ-REF	Object reference data
POINTER	Pointer data

The following symbols are used to display file types and access methods for FD items.

SSAM	Sequential file, sequential access
LSAM	Line sequential file, sequential access
RSAM	Relative file, sequential access
RRAM	Relative file, random access
RDAM	Relative file, dynamic access
ISAM	Indexed file, sequential access
IRAM	Indexed file, random access
IDAM	Indexed file, dynamic access
PSAM	Presentation file, sequential access

- [8] Starting point

  Indicates the base registers and base points to which data items are assigned.

- [9] Number of dimensions

  Displays the number of dimensions for indexing or subscripting.

- [10], [11] Program name

  Displays the name of a program as a delimiter if the program is nested.

  For class definitions, however, each definition and delimiter is displayed in the following format:

```
**Class name**
** FACTORY **
** OBJECT **
** MET(method-name)**
```

- [12] Encoding form

  Displays the encoding forms of the following data items:

  - ALPHANUM (Alphanumeric)
  - AN-EDIT (Alphanumeric edited)
  - NATIONAL (National)
  - NAT-EDIT (National edited)

  Displays the encoding forms using the following symbols:

- SJIS : Shift JIS
- UTF8 : UTF-8
- UTF16LE : UTF-16 little endian
- UTF16BE : UTF-16 big endian
- UTF32LE : UTF-32 little endian
- UTF32BE : UTF-32 big endian

Program control information listing output format

```
    ADDR              FIELD-NAME                      LENGTH
 [1]
    ** GWA **
      * GCB *
data+00000000          GCB FIXED AREA                     8

      * GMB *
data+00000008          GMB POINTERS AREA                384
data+00000008            VNAL                           128
     ........            VNAS                             0
data+00000088            VNAO                           256
     ........            FARA                             0
     ........            METHOD ADDRESS AREA              0
     ........            BEA                              0
     ........            BEAD / BEAR                      0
     ........            FMBE                             0
     ........          GMB CONTROL BLOCKS AREA            0
     ........            STRONG TYPE AREA                 0
     ........          FAI AREA                           0
     ........            ATTR INFO FIXED AREA             0
     ........            REUSE INFO TABLE                 0
     ........            METHOD TABLE                     0
     ........            INTERFACE TABLE                  0
     ........            METHOD COPY TABLE                0
     ........          IAI AREA                           0
     ........            ATTR INFO FIXED AREA             0
     ........            REUSE INFO TABLE                 0
     ........            METHOD TABLE                     0
     ........            INTARFACE TABLE                  0
     ........            METHOD COPY TABLE                0
     ........          INVOKE PARM INFO                   0
     ........          AS MODIFY INFO                     0

    ** COA **
      * CCB *
rdat+00000000          CCB FIXED AREA                     4
     ........          STANDARD CONSTANT AREA             0

      * CMB *
     ........          CMB POINTERS AREA                  0
     ........            BEAI                             0
     ........            BVAI                             0
     ........            IPA                              0
rdat+00000010          LITERAL AREA                      40
rdat+00000040          CONSTANT SECTION DATA             24
rdat+00000058          CMB CONTROL BLOCKS AREA          816
rdat+00000058            FMB2                           224
     ........            FMBI                             0
     ........            SMB1                             0
     ........            SMB2                             0
     ........            EDT                              0
```

```
     ........           EFT                            0
rdat+00000138           FMB2 ADD-PRM LIST             16
     ........           ERROR PROCEDURE                0
     ........           ALPHABETIC NAME                0
     ........           CLASS NAME                     0
     ........           CLASS TEST TABLE               0
     ........           TRANS-TABLE PROTO              0
     ........           CIPB                           0
     ........           DOG TABLE                      0
     ........           EXTERNAL DATA NAME             0
     ........           NATIONAL-MSG F-NAM             0
     ........           FLOW BLOCK INFO                0
     ........           SQL AREA                       0
     ........           SPECIAL REGISTERS              0
rdat+00000148           EPA CONTROL AREA             576
     ........           SCREEN CONTROL AREA            0
     ........           EXCEPTION PROC LIST            0
     ........           SQL INIT INFO                  0
     ........           OLE PARM INFO                  0
     ........           CALL PARM INFO                 0
     ........           CALL PARM ADD INFO             0
     ........           UWA RECORD INFO                0
     ........           UWA INFO                       0
     ........           UWA RECORD LIST                0
     ........           RECORD INFO                    0
     ........         CALL PARM ADDRESS LIST           0
     ........         FCM FMB1 OFFSET LIST             0
     ........         FCM OBJ-REF OFFSET LIST          0
     ........         ICM FMB1 OFFSET LIST             0
     ........         ICM OBJ-REF OFFSET LIST          0
     ........         MCM AREA / OBJ-REF LIST          0
     ........         CFOR OFFSET LIST                 0
     ........         CLASS NAME INFO                  0
     ........         AS MODIFY / PARAM INFO           0
     ........         METHOD NAME INFO                 0
     ........         INIT TABLE                       0

    ** HGWA **
      * HGCB *
heap+00000000        HGCB FIXED AREA                  72

      * HGMB *
heap+00000048        LIA FIXED AREA                  256
     ........         IWA1 AREA                        0
     ........           LINAGE COUNTER                 0
     ........           OTHER AREA                     0
     ........           ALTINX                         0
     ........           PCT                            0
heap+00000148        LCB AREA                        160
heap+000001E8        HGMB POINTERS AREA               88
     ........           VPA                            0
     ........           PSA                            0
heap+000001E8           BVA                          24
     ........           BEA                            0
     ........           FMBE                           0
heap+00000200           CONTROL ADDRESS TABLE         64
     ........           MUTEX HANDLE AREA              0
heap+00000240        HGMB CONTROL BLOCK AREA         384
heap+00000240           FMB1                         384
     ........           SMB0                           0
     ........           SPECIAL REGISTERS              0
     ........           S-A LIST                       0
     ........           DPA                            0
```

```
     ........            SQL CONTROL AREA                  0
     ........            DMA                               0
     ........            SCREEN CONTROL AREA               0
     ........            METHOD INIT INFO                  0
     ........          CFOR                                0

      * HGWS *
heap+000003C0          DATA AREA                          12

    ** STK **
      * SCB *
stac+00000000          ABIA                               24
stac+00000020          SCB FIXED AREA                    112
stac+00000090          TL 1ST AREA                        48
stac+000000C0          LCB AREA                          144
stac+00000150          SGM POINTERS AREA                   8
     ........            VPA                               0
     ........            PSA                               0
     ........            BVA                               0
stac+00000150            BHG                               8
     ........            BOD                               0
stac+00000158          LIA VARIABLE AREA                 160
     ........          SOR AREA                            0
     ........          TSG AREA                            0
stac+000001F8          TRG AREA                            8
     ........          SGM CONTROL BLOCKS AREA             0
     ........            FMB1                              0
     ........            SMB0                              0
     ........            SPECIAL REGISTER                  0
     ........            MIA                               0
     ........            SQL CONTROL AREA                  0
stac+00000200          IWA3 AREA                          56
     ........            ALTINX                            0
     ........            USESAVE                           0
stac+00000200            ENTSAVE                           8
     ........            PCT                               0
     ........            CONTENT                           0
     ........            USEOSAVE                          0
stac+00000208            RTNADDR                          16
     ........            RTNAREA                           0
     ........            LINAGE COUNTER                    0
     ........            OTHER AREA                        0
stac+00000218            PARM                             32
     ........            CALL PARM INFO                    0
     ........          DATA AREA                           0
stac+00000240          TL 2ND AREA                        96
     ........          PRESERVED REGISTER                  0
stac+000002A0          SCRATCH / REGISTER PARM            96
     ........          RTS RETURN AREA                     0
stac+00000308          LIA FIXED ADDR                      8
stac+00000310          CBL STRING                          8
stac+00000318          RESERVED REGISTER                  48
stac+00000348          RETURN ADDRESS                      8
     ........          PARM AREA                           0

    ** LITERAL-AREA **
    ADDR               0 . . .  4 . . .  8 . . .  C . . .    0123456789ABCDEF
rdat+00000010          01000000 08000000 20000000 10000000   ........ ........
rdat+00000020          40000000 00000000 5359534F 55542020   @.......SYSOUT
rdat+00000030          80000000 00000000                      ........
```

Explanation of the program control listing:

- [1] Indicates the locations of work areas and data areas assigned in the target program.

- [2] Indicates the constant areas in the target program.

Section size listing output format

```
      ** PROGRAM SIZE **
[1]
    .TEXT SIZE         =      10656 bytes
    .DATA SIZE         =        440 bytes
     ** EXECUTION DATA SIZE **
[2]
    HEAP SIZE          =        976 bytes
    STACK SIZE         =       1168 bytes
```

Explanation of the section size listing

- [1] Displays the sizes of the .text and .data sections in the target program.
- [2] Displays the sizes of areas required for execution.

  For class definitions, however, the sizes are displayed in the following format:

```
     ** EXECUTION DATA SIZE **
Class name
     HEAP SIZE         =         0 byte
Method name                                          --+
     STACK SIZE        =       384 bytes               --+  [3]
           :
```

- [3] The stack size is displayed for each method definition.

3.2 Program Structure

This section explains the following:

- The structure of the executable program created by compiling and linking the source program.
- The type of linkage.
- The structure of the executable program generated from linkage.
- How to link to the source program to create the executable program.

3.2.1 Overview

The executable program in this product has been created by linking the following programs to the re-locatable program generated by the COBOL compiler:

- Startup routine
- Runtime library subroutine for COBOL.
- Runtime library subroutine for C language.

These programs are automatically linked when linking with the cobol command. However, if you link the program with the ld command, the ld command must be used.

3.2.2 Linkage Type and Program Structure

Linkages can be static or dynamic linked.

Static Linkage

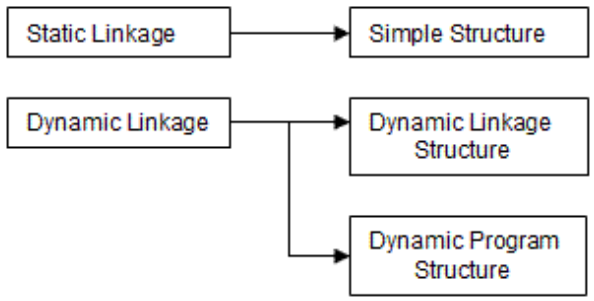
When programs are statically linked, a group of modules is linked into a single executable file.

Dynamic Linkage

When programs are dynamically linked, called programs are linked with calling programs at execution time.

The program structure created in each linkage method is shown in Figure below. They are explained below the image. Hereafter, main program refers to a program that starts first and subprogram(s) refers to those called by the main program. Also note that subprograms can be called by other subprograms.

Figure 3.1 Linkage Type and Program Structure

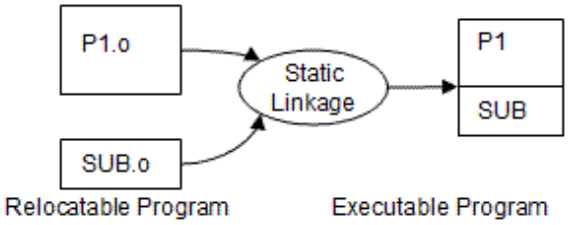


Simple Structure

In a simple structure, more than one re-locatable program is statically linked into a single executable program. Thus, all of the main program and subprograms are loaded into virtual memory at the start of execution, giving highly efficient subprogram invocation. However, to create a simple structure executable file, all the subprograms are required at link time.

The following figure shows an overview of the simple structure:

Figure 3.2 Simple Structure



Dynamic Link Structure

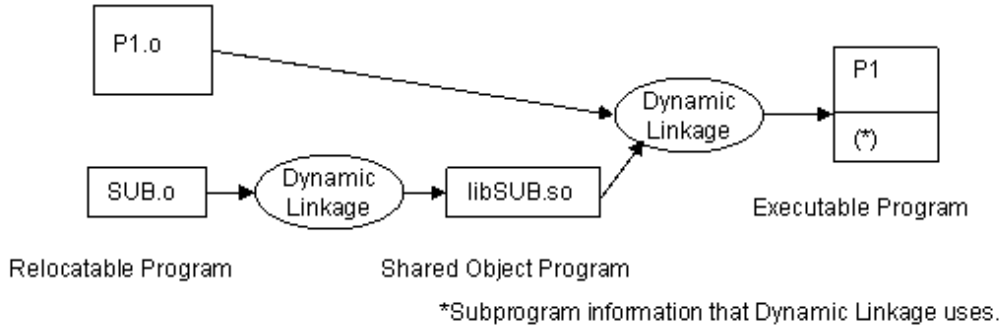
A dynamic link structure is an executable program created by dynamically linking the re-locatable program of the main program and the shared object program of a subprogram.

Unlike the simple structure, the executable file does not contain a subprogram in the dynamic link structure. The subprogram is loaded into virtual memory by the dynamic linker when needed.

The dynamic linker of the system performs loading by using the subprogram information created in the executable file during dynamic linkage and the path list set in environment variable LD_LIBRARY_PATH. When you move the directory where the subprogram is stored after linkage, the destination path is required to have been set in environment variable LD_LIBRARY_PATH.

The following figure shows the overview of the dynamic link structure:

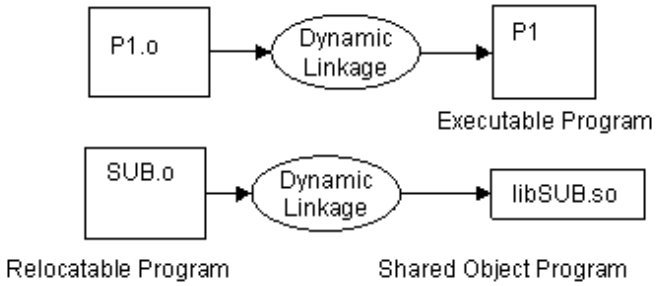
Figure 3.3 Dynamic Link Structure



Dynamic Program Structure

In a dynamic program structure, only the main program's re-locatable program is an executable program with dynamic linkage. Unlike the dynamic link structure, the dynamic linker's subprogram information is not included in the executable program. The shared object program of the subprogram is executed once it is needed. In the dynamic program structure, the subprogram is loaded by being asked to the COBOL runtime system from the invoking program. In this case, the system loader is used. The following figure is an overview of the dynamic program structure:

Figure 3.4 Dynamic Program Structure



Program Structure and CALL Statement

The program structure is determined by the format of the CALL statement, options specified when compiling, and the linkage type. The following table lists the relationships among the program structure, CALL statement, compiler options, and linkage type. For information on the compiler option DLOAD, see "Appendix A Compiler Options".

Table 3.2 Program Structure, CALL Statement, Compiler Options Relationships.

		Compiler Option	
		DLOAD	NODLOAD or omitted
CALL statement coding method	CALL "program-name"	Dynamic program structure	Simple structure or dynamic link structure (*1)
	CALL data-name	Dynamic program structure	Dynamic program structure
	CALL "program-name" used together with CALL data-name	Dynamic program structure	Dynamic link structure (*2) Dynamic program structure (*3)

*1 : The shared object programs to be called for linkage are required in the dynamic link structure.

*2 : The dynamic link structure is used for calling with a program name specified.

*3 : The dynamic link structure is used for calling with a data name specified.

The dynamic program structure usually requires entry information. For details on entry information, see "4.1.3 Subprogram entry information".

Relationship between the program structure and CANCEL statement

The CANCEL statement initializes the status of program called for the second time or subsequently. However, whether or not the CANCEL statement can be used depends on the program structure. The table below shows what program structures allow the CANCEL statement to be used.

Table 3.3 Relationship 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 the internal program, see "8.2.7 Internal Programs".

3.2.3 Creating an Executable Program with a Simple Structure

The following is an example of creating an executable program with a simple structure.

Example

Compile and link the three source programs: P1, P2, and P3. The relationship between the programs is as follows:

- P1 calls P2 and P3.
- P2 and P3 call no programs.

How to Create an Executable Program With Only the cobol Command

```
$  cobol -c P2.cob P3.cob                                [1]
HIGHEST SEVERITY CODE = I
HIGHEST SEVERITY CODE = I
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=2
$  cobol -dn -M -o P1 P1.cob P2.o P3.o                   [2]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

[1] Compile the subprogram to create a re-locatable program.

- Input

  P2.cob (source file)

  P3.cob (source file)

- Output

  P2.o (object file)

  P3.o (object file)

- Option

  -c (specifying to compile only)

[2] Compile and link the main program to create an executable program.

- Input

  P1.cob (source file)

  P2.o (object file)

  P3.o (object file)

- Output

  P1 (executable file)

- Option

  -dn (specifying the static linkage)

  -M (specifying the main program)

  -o (output destination of the executable program)

How to Link the Subprogram as an Archive Library Subroutine

```
$ cobol -c P2.cob P3.cob                                [1]
HIGHEST SEVERITY CODE = I
HIGHEST SEVERITY CODE = I
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=2
$ ar r libP0.a P2.o P3.o                                [2]
$ cobol -dn -M -o P1 -L. -lP0 P1.cob                    [3]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

[1] Compile the subprogram to create a re-locatable program.

- Input

  P2.cob (source file)

  P3.cob (source file)

- Output

  P2.o (object file)

  P3.o (object file)

- Option

  -c (specifying to compile only)

[2] Create an archive library subroutine of the subprogram.

- Input

  P2.o (object file)

  P3.o (object file)

- Output

  ibP0a (archive library subroutine)

- Option

  -r (specifying an archive library subroutine)

[3] Compile and link the main program to create an executable program.

- Input

  P1.cob (source file)

  libP0.a (archive library subroutine)

- Output

  P1 (executable file)

- Option

  -dn (specifying the static linkage)

  -M (specifying the main program)

-o (output destination of the executable program)

-l (Library to be linked)

How to Link with the ld Command

For information on how to link with the ld command, see "K.2.2 How to use the ld command by program structure".

3.2.4 Creating an Executable Program with a Dynamic Link Structure

The following is an example of creating an executable program with a dynamic link structure.

Example

Compile and link the three source programs: P1, P2, and P3. The relationship between the programs is as follows:

- P1 calls P2 and P3.
- P2 and P3 call no programs.

When Creating P2 and P3 as a Separate Shared Object Program

```
$ cobol -dy -shared -o libP2.so P2.cob               [1]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ cobol -dy -shared -o libP3.so P3.cob               [2]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ cobol -dy -M -o P1 -L. -lP2 -lP3 P1.cob            [3]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

[1] Compile and link the subprogram to create a shared object program.

- Input

  P2.cob (source file)

- Output

  libP2.so (shared object file)

[2] Compile and link the subprogram to create a shared object program.

- Input

  P3.cob (source file)

- Output

  libP3.so (shared object file)

[3] Compile and link the main program to create an executable program.

- Input

  P1.cob (source file)

  libP2.so (shared object file)

  libP3.so (shared object file)

- Output

  P1 (executable file)

When Creating P2 and P3 as One Shared Object Program

```
$ cobol -dy -shared -o libP0.so P2.cob P3.cob        [1]
HIGHEST SEVERITY CODE = I
HIGHEST SEVERITY CODE = I
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=2
```

```
$ cobol -dy -M -o P1 -L. -lP0 P1.cob                    [2]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

[1] Compile and link the subprogram to create a shared object program.

- Input

  P2.cob (source file)

  P3.cob (source file)

- Output

  libP0.so (shared object file)

[2] Compile and link the main program to create an executable program.

- Input

  P1.cob (source file)

  libP0.so (shared object file)

- Output

  P1 (executable file)

How to Link with the ld Command

For information on how to link with the ld command, see "K.2.2 How to use the ld command by program structure".

3.2.5 Creating an Executable Program with a Dynamic Program Structure

The following is an example of creating an executable program with a dynamic link program.

Example

Compile and link the three source programs: P1, P2, and P3. The relationship between the programs is as follows:

- P1 calls P2 and P3.
- P2 and P3 call no programs.

In the dynamic program structure, the subprogram is loaded by the COBOL runtime system. At this time, you must specify entry information so that the COBOL runtime system can identify the shared object program to be loaded. However, if the shared object program has the name shown below, you do not have to specify entry information. For details about entry information, refer to "4.1.3 Subprogram entry information". The COBOL runtime system loads the shared object program with the following name:

```
libXXX.so
```

XXX is a program name specified in the CALL statement.

When Creating P2 and P3 as a Separate Shared Object Program

```
$ cobol -dy -shared -o libP2.so P2.cob                  [1]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ cobol -dy -shared -o libP3.so P3.cob                  [2]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ cobol -dy -M -o P1 -WC,"DLOAD" P1.cob                 [3]
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

[1] Compile and link the subprogram to create a shared object program.

- Input

  P2.cob (source file)

- Output

  libP2.so (shared object file)

[2] Compile and link the subprogram to create a shared object program.

- Input

  P3.cob (source file)

- Output

  libP3.so (shared object file)

[3] Compile and link the main program to create an executable program.

- Input

  P1.cob (source file)

- Output

  P1 (executable file)

How to Link with the ld Command

For information on how to link with the ld command, see "K.2.2 How to use the ld command by program structure".

Notes on Dynamic Program Structure

If you do not specify either of the following using the dynamic program structure, you cannot call several subprograms that have been created as one shared object program. Accordingly, in these examples, you must create P2 and P3 as separate shared object programs.

There are two ways to call several programs that have been created as one shared object program. These are as follows:

- Specify the entry information file name in the CBR_ENTRYFILE environment variable.
- Link the shared objects called in the executable program using the -l option. However, note that the CANCEL statement for programs called by this method is invalid.

For details regarding the calling of subprograms using dynamic program configuration, refer to "Chapter 8 Calling Subprograms (Inter-Program Communication)".

3.3 cobol Command

The cobol command compiles and links a COBOL source program and creates a re-locatable program, shared object program, and executable program. The following explains the cobol command format:

```
$  cobol [compiler-options and link-options]  File-name...
```

Descriptions of Options and Operands

There must be at least one tab or blank space between the command name and the operands.

If the options are specified in the environment variable COBOLOPTS they will be specified every time you use the cobol command.

```
$  COBOLOPTS="-Dt -WC,LINESIZE(80),MESSAGE" ; export COBOLOPTS
$  cobol -M p1.cob
```

The above example is equivalent to the following example:

```
$  cobol -Dt -WC,"LINESIZE(80),MESSAGE" -M p1.cob
```

In the descriptions to follow, file and directory names can be specified with either an absolute path or a relative path name.

Compiler Options

Compiler options specifies a variety of information to be sent to the COBOL compiler. For the content of specification, see "3.3.1 Compiler Options".

Link Options

Link options specifies a variety of information to be sent to the linker. For the content of specification, see "3.3.2 Link Options".

File Name

Specifies the path name to the file where the source program is stored (source file), or the path name where the re-locatable program is stored (object file). More than one file can be specified.

Note that for re-locatable and shared object programs, compilation will not take place, only linkage will take place.

Note

- If the total number of parameters (compilation-related options, link-related options, and file names) specified in the cobol command exceeds 4000, the cobol command may terminate abnormally. In such case, reduce the number of parameters specified in the cobol command to 4000 or less.
- Write permission for the current directory is required.

3.3.1 Compiler Options

Table 3.4 Compiler options

Options That Relate to Compile Resources	-dr	Specification of input/output destination directory of the repository file
	-I	Specification of library file directory
	-m	Specification of form descriptor file directory
	-R	Specification of repository file input destination directory
	-ds	Specification of source analysis information file output directory
Options That Relate to Compile Listings	-dp	Specification of compile list file directory
	-P	Specification of the file name of the compile list
Options That Relate to Object Program Generating	-do	Specification of the object file directory
	-M	Specification for the main program compilation
	-Tm	Specification for multithreaded program compilation
Options That Relate to Debugging Functions	-Dc	Specification for using COUNT function
	-dd	Specification for the debugging information file directory
	-Dk	Specification for using CHECK function
	-Dr	Specification for using TRACE function
	-Dt	Specification for using the remote debug function of NetCOBOL Studio
Others	-c	Specification for compilation only
	-i	Specification of option file
	-v	Specification for type of information to be output
	-WC	Specification of compiler options

Note

If you specify the same option more than once, the most recent specification will be enabled unless otherwise specified.

The compiler options are described below, listed in alphabetical order.

3.3.1.1 -c (Specification for compilation only)

Specifies when you compile but do not link the program.

```
-c
```

3.3.1.2 -Dc (Specification for using COUNT function)

```
-Dc
```

To use the COUNT function, specify the -Dc option. For information on the COUNT function, see "5.4 Using the COUNT Function".

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. For more information, see "A.2.8 COUNT (whether the COUNT function should be used)".

3.3.1.3 -Dk (Specification for using CHECK function)

```
-Dk
```

To use the CHECK function, specify the -Dk option. For information on the CHECK function, see "5.3 Using the CHECK Function". The -Dk option specifies that code for inspecting subscripts, indexes, and sub-references is incorporated into the object program, consequently degrading the execution performance. It is recommended to omit the -Dk option when you use the cobol command.

Information

The -Dk option is identical to the compiler option CHECK. For more information, see "A.2.5 CHECK (whether the CHECK function should be used)".

3.3.1.4 -Dr (Specification for using TRACE function)

```
-Dr
```

To use the TRACE function, specify the -Dr option. For information on the TRACE function, see "5.2 Using the TRACE Function". The -Dr option specifies that code for displaying trace information is incorporated into the object program, consequently degrading the execution performance. It is recommended to omit the -Dr option when you use the cobol command.

Information

The -Dr option is identical to the compiler option TRACE. For more information, see "A.2.50 TRACE (whether the TRACE function should be used)".

3.3.1.5 -dr (Specification of input/output destination directory of the repository file

```
-dr directory
```

Use the -dr option if the repository files are stored in or should be written to a different directory from the source file.

The -dr option is effective only in the compilation of the class definition.

If the -dr option is not specified, the repository files are created in the directory containing the source file.

The directory specified with the -dr option is also used as a destination directory when deriving external repositories.

3.3.1.6 -ds (Specification of output directory of the source analysis information file)

```
-ds
```

To change the source analysis information file storage directory, specify the new directory in the -ds option.

The -ds option has no effect unless the compiler option SAI is specified.

If the -ds option is omitted, the source analysis information file is created in the directory containing the source files.

Information

The -ds option is equivalent to the SAI compile option. For details, see "A.2.37 SAI (whether a source analysis information file is output) ".

Note

If the NOSAI compile option is enabled, the -ds option is not valid.

3.3.1.7 -Dt (Specification for using the remote debug function of NetCOBOL Studio)

```
-Dt
```

To use the remote debug function of NetCOBOL Studio, specify the -Dt option. For information on the remote debug function of NetCOBOL Studio, see "Chapter 18 Using the Debugger".

Information

The -Dt option is identical to the compiler option TEST. For more information, see "Chapter 18 Using the Debugger" and "A.2.48 TEST (whether the remote debug function of NetCOBOL Studio should be used)".

3.3.1.8 -dd (Specification for the debugging information file directory)

```
-dd directory
```

Use the -dd option to specify the directory where the debugging information is stored. If the -dd option is not specified the debugging information file is created in the directory containing the source file. The -dd option has no effect unless the -Dt option or compiler option TEST is specified.

Information

"A.2.48 TEST (whether the remote debug function of NetCOBOL Studio should be used)"

3.3.1.9 -do (Specification of the object file directory)

```
-do directory
```

Use the -do option to store the object file in a different directory from the source file. If the -do option is not specified, the object file is created in the directory containing the source file. The -do option has no effect unless compiler option OBJECT is specified.

Information

"A.2.32 OBJECT (whether an object program should be output)"

3.3.1.10 -dp (Specification of compile list file directory)

```
-dp directory
```

Use the -dp option to store the compile list file in a different directory from the source file. The -dp option has no effect unless the -P option is specified.

When the -dp option is specified, the compiler list is as follows:

- When the file name is specified to the -P option, the compiler list file name becomes a name that unites "Directory name specified to the -dp option" with "File name specified to the -P option"

  Example: The file name is specified to the -P option

  ```
  cobol -P out.lst -dp /tmp/test cobtest.cob
  ```

  -> /tmp/test/out.lst

- When - is specified to the -P option, the compiler list file name becomes a name that unites "Directory name specified to the -dp option" with "Source-file-name.lst"

  Example: - is specified to the -P option

  ```
  cobol -P- -dp /tmp/test cobtest.cob
  ```

  -> /tmp/test/cobtest.lst

Information

"3.3.1.15 -P (Specification of the file name of the compile list)"

3.3.1.11 -I (Specification of library file directory)

```
-I directory
```

To use the COPY statement in source programs, use the -l option to specify the directory where the library files written in the COPY statement of the source program are stored. If you specify more than one -l option, the library files are searched for in the order the -l options are specified. If the directory specified does not contain the library files to search, the current directory will be searched.

3.3.1.12 -i (Specification of option file)

```
-i file
```

When you specify the compiler options with an option file (containing the compiler options character strings), specify the option file name. Use a text editor to create the option file. The content of the option file is identical to the compiler option list specified with the -WC option. The following is an example of the option file:

```
MESSAGE,NUMBER,OPTIMIZE
```

3.3.1.13 -M (Specification for the main program compilation)

```
-M
```

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

The -M option is identical to the compiler option MAIN. For more information, see "A.2.23 MAIN (main program/sub-program specification)".

3.3.1.14 -m (Specification of form descriptor file directory)

```
-m directory
```

To use the COPY statement with IN/OF XMDLIB specified to incorporate the record definition from the form descriptor, use the -m option to specify the directory where the form descriptor files are stored. If you specify more than one -m option, the form descriptor files are searched for in the order the -m options are specified. If the directory specified does not contain the form descriptor files searched for, the current directory will be searched.

3.3.1.15 -P (Specification of the file name of the compile list)

```
-P file
```

When you store the compile list in a file, specify the name of the file. For details on the compile list options, see "3.1.7 Information Provided by the NetCOBOL Compiler" in this chapter. If you specify the -P option more than once, the file name specified last will be enabled.

If you want to output the compile list file in the form of "source file name.lst", specify a hyphen (-) instead of the file name.

Note

The compile list will be output based on the following directory:

- If the -dp option is specified at the same time

  The directory specified in the -dp option

- If the -dp option is not specified & If the file name is specified with the -P option

  The current directory

- If the -dp option is not specified & If a hyphen (-) is specified with the -P option

  The directory where the source file is stored

3.3.1.16 -R (Specification of repository file input destination directory)

```
-R directory
```

To use external repositories specified in the REPOSITORY paragraph, use the -R option to specify the directory where the repository files are stored. If the repository files are in a number of directories, specify the -R option for each directory. Directories are searched in the order the -R options are specified.

3.3.1.17 -Tm (Specification for multithreaded program compilation)

```
-Tm
```

Specify this option when you compile a multithreaded program. For the details of the multithreaded model program, see "Chapter 16 Multithread Programs".

The -Tm option is identical to the compiler option THREAD(MULTI).

3.3.1.18 -v (Specification for type of information to be output)

```
-v
```

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

- Cobol command version information
- Command line character string when the ld command is called

Note

- If -c is specified at the same time, the command line character string of the Id command is not output.
- If -v is specified without any other options, resource file names, object file names, or similar, the cobol command outputs only version information and terminates normally. In this case, the cobol command return value is 0.

Information

"3.3.3 Return Values of the cobol Command"

3.3.1.19 -WC (Specification of compiler options)

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

Specifies the compiler options. Separate multiple compiler options with commas (,). If a more than one compiler option is specified, the last-specified setting of the option is the one that is used. For information on the content and specification format of the compiler options, see "Appendix A Compiler Options".

The following are the compiler option specification methods and their priorities:

1. Compiler options specified in the compiler statement in the source program.
2. Compiler options specified with the -WC option of the cobol command.
3. Compiler options specified with the -WC option of the environment variable COBOLOPTS.
4. Options specified in the cobol command.
5. Options specified in the environment variable COBOLOPTS.
6. Compiler options in the options file specified with the -i option of the cobol command.

3.3.2 Link Options

Table 3.5 Link options

-dy/-dn	Specification of linkage mode
-G/-shared	Specification for creating a shared object program
-L	Specification of adding a library search path name
-l	Specification of the subprogram or library to link
-o	Specification of the object file
-Tm	Specification for linking a multithreaded program
-Wl	Specification of link options

The link options are described below, listed in alphabetical order.

3.3.2.1 -dy/-dn (Specification of linkage mode)

```
-dy
```

or

```
-dn
```

Specifies whether the library containing the subprogram called from the object program (-dn) is to be statically linked or dynamically linked (-dy). When the option is omitted, -dy is assumed to have been specified, which executes the dynamic linkage.

-dy is specified when dynamic linkage and static linkage are to be done at the same time. Specify the static linkage subprogram at the end of the -Wl option.

3.3.2.2 -G/-shared (Specification for creating a shared object program)

```
-shared
```

or

```
-G
```

Specify this option when creating a shared object program. -G is equivalent to -shared.

3.3.2.3 -L (Specification of adding a library search path name)

```
-L directory
```

Specify this option when you want to add the directories in which to search for libraries.

3.3.2.4 -l (Specification of the subprogram or library to link)

```
-l name
```

When creating a dynamic link structure, use the name of the shared object library of the subprogram called from the object program for "name". When using a variety of COBOL functions, use a necessary library name for "name". With this option, shared object programs with the name "lib name.so" or archive library subroutines with the name "lib name.a" are sequentially searched from the following directories:

- Directory specified in the -L option of the cobol command

More than one option can be specified. If you do so, they are searched in the order specified.

3.3.2.5 -o (Specification of the object file)

```
-o file
```

Specifies the file that stores the executable program or shared object program. If the -o option is omitted, the program will be stored in a.out. When creating a subprogram as a shared object program, specify the file name "lib subprogram.so" with the -o option.

3.3.2.6 -Tm (Specification for linking a multithreaded program)

```
-Tm
```

Specify this option when you automatically link the library required for a multithreaded program. For the details of the multithreaded model program, see "Chapter 16 Multithread Programs".

3.3.2.7 -Wl (Specification of link options)

```
-Wl,"link option"
```

Specifies the option that indicates to the ld command. When two or more -Wl options are specified, the last -Wl option is effective.

For the information on the content and specification format of the options specified in the ld command, see "Appendix K ld Command" and the ld command manuals.

3.3.3 Return Values of the cobol Command

The cobol command's return values are determined by the highest severity code at program compilation. The following table shows what severity code produces what return value:

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

When you create an executable program with the cobol command, the command internally runs the ld command. If an error occurs during execution of the ld command, compare the above values with the return value of the ld command and the larger is the cobol command's return value.

Chapter 4 Executing Programs

This chapter explains how to execute a COBOL program. It includes instructions for setting up the Runtime Environment information, invoking an application program, managing runtime error messages and handling program termination. Additional information is provided for managing stack space and virtual memory.

4.1 Setting up the Runtime Environment Information

This section explains how to set up the runtime environment information.

4.1.1 Runtime Environment

The runtime environment is information required to run a COBOL application. An environment variable is information that specifies a file-identifier. For the details on the environment variables, see "Appendix E Environment Variable List".

Before executing COBOL programs, you need to have the environment variables set for each function. For the functions and how to set their environment variables, see the explanation for each function in "Appendix E Environment Variable List".

Note that the environment variables should not match those used by other utilities and COBOL programs. Also, do not use environment variables starting with SYS, CBR, or COB that are reserved by the COBOL runtime system.

The environment variables available at execution of the COBOL program are as follows:

Runtime Environment-related

- CBR_CBRFILE
- CBR_CBRINFO
- CBR_CONVERT_CHARACTER
- GOPT
- MGPRM

See "Appendix E Environment Variable List".

Invoking Subprograms-related

- CBR_ENTRYFILE
- LD_LIBRARY_PATH

See "Chapter 8 Calling Subprograms (Inter-Program Communication)".

File Processing-related

- File-identifier
- CBR_INPUT_BUFFERING
- CBR_CLOSE_SYNC
- CBR_TRAILING_BLANK_RECORD
- CBR_FILE_USE_MESSAGE
- CBR_EXFH_API
- CBR_EXFH_LOAD
- CBR_FILE_BOM_READ

- CBR_FILE_SEQUENTIAL_ACCESS

See "Chapter 6 File Processing".

Presentation File-related

- File-identifier
- MEFTDIR

Print File-related

- File-identifier
- FCBDIR
- CBR_FCB_NAME
- CBR_PRT_INF
- FOVLDIR
- CBR_LP_OPTION
- CBR_PRINTFONTTABLE

See "Chapter 7 Printing".

Sort-merge-related

- BSORT_TMPDIR

See "Chapter 10 Using SORT/MERGE Statements (Sort-Merge Function)".

ACCEPT/DISPLAY-related

- Name specified in compiler option SSIN or SSOUT
- CBR_JOBDATE
- CBR_MESSOUTFILE
- CBR_COMPOSER_SYSOUT
- CBR_COMPOSER_SYSERR
- CBR_COMPOSER_CONSOLE
- CBR_DISPLAY_CONSOLE_OUTPUT
- CBR_DISPLAY_SYSOUT_OUTPUT
- CBR_DISPLAY_SYSERR_OUTPUT
- CBR_DISPLAY_CONSOLE_SYSLOG_LEVEL
- CBR_DISPLAY_SYSOUT_SYSLOG_LEVEL
- CBR_DISPLAY_SYSERR_SYSLOG_LEVEL
- CBR_DISPLAY_CONSOLE_SYSLOG_IDENT
- CBR_DISPLAY_SYSOUT_SYSLOG_IDENT
- CBR_DISPLAY_SYSERR_SYSLOG_IDENT

See "Chapter 9 Using ACCEPT and DISPLAY Statements" and "D.3 Obtaining the Year Using the CURRENT-DATE Function".

Object-oriented-related

- CBR_CLASSINFFILE

- CBR_INSTANCEBLOCK

See "Chapter 15 Developing and Executing Object-Oriented Programs".

Multithread-related

- CBR_THREAD_TIMEOUT
- CBR_SYMFOWARE_THREAD
- CBR_SYSERR_EXTEND
- CBR_SSIN_FILE

See "Chapter 16 Multithread Programs".

Debugging Functions-related

- SYSCOUNT
- CBR_TRACE_FILE
- CBR_TRACE_PROCESS_MODE
- CBR_MEMORY_CHECK

See "Chapter 5 Debugging Programs".

Remote Debug Function of NetCOBOL Studio

- CBR_ATTACH_TOOL

See "18.1.4 CBR_ATTACH_TOOL".

Code-related

- LANG
- LC_ALL

See "Appendix J National Language Code".

Intrinsic Functions-related

- CBR_FUNCTION_NATIONAL

See "Appendix D Intrinsic Function List".

Output of message when executing-related

- CBR_MESS_LEVEL_CONSOLE
- CBR_MESS_LEVEL_SYSLOG
- CBR_MESSOUTFILE
- CBR_COMPOSER_MESS

See "4.3 Specifying How to Output Execution Time Messages".

CSV format data operation

- CBR_CSV_OVERFLOW_MESSAGE
- CBR_CSV_TYPE

See "Chapter 21 Operation of CSV (Comma Separated Value) data".

Others

- TMPDIR

4.1.2 How to Set Runtime Environment Information

Runtime environment information can be set as follows:

a. By setting in the shell initialization file.

b. By using environment variable setup commands.

c. By setting in the runtime initialization file.

d. By setting from the command line (runtime option)

The environment variables in the runtime initialization file are reflected on those of an application during execution of the COBOL program.

It is recommended that runtime environment information is set by (a) or (b).

- The methods (a) and (b) require different settings, depending on the shell being used. See the appropriate documentation of the shell being used as to how to set the runtime environment information.
- The runtime environment information specified in the runtime initialization file affects the runtime performance because it is fetched when opening the runtime environment of the COBOL program. Therefore, it is recommended as follows:
  - Set the environment variable information to the user environment variable with a shell command before starting the program.
  - Set only the information required for the program to be executed in the runtime initialization file.
- When operating with the Interstage Application Server CORBA work unit, set the environment of the work unit and set the environment variable.

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.

4.1.2.1 How to Set in the Shell Initialization File

This method is used for setting environment variable information using the shell initialization files common to the system or specific to users. This is useful if you want to have the values of the environment variables common to more than one application defined.

4.1.2.2 How to Set by Using Environment Variable Setup Commands

This method is used for setting environment variable information using environment variable setup commands of a shell or shell program. The environment variable information, which is set from the shell, becomes valid in programs started from that shell. Using a shell program enables the performance of all steps from setup to execution to be performed in one step. The environment variable information, which is set from the shell program, becomes valid only in programs started from that shell file.

4.1.2.3 How to Set in the Runtime Initialization File

This section explains how to create a runtime initialization files before execution of the program and set the runtime environment information. Runtime initialization file saves the information for executing programs created in COBOL and is used for executing programs. The "COBOL.CBR" of the directory containing the executable program is normally handled as the runtime initialization file. When the executable program is invoked with a specified path, "COBOL.CBR" of the directory containing the executable program is handled as the runtime initialization file.

See the following if a file created with a name other than "COBOL.CBR" is to be handled as the runtime initialization file:

- To specify the runtime initialization file with environment variable CBR_CBRFILE, see "Appendix E Environment Variable List".
- To specify the file with the command line option, see "4.2 Execution Procedure".

The program can be executed without the runtime initialization file.

4.1.2.3.1 Runtime initialization file contents

Write the environment variable information common to each program in the runtime initialization file. The environment variable information described here is valid until the application is terminated.

The contents of the runtime initialization file are shown below:

```
; Comment                                              ... [1]
environment-variable-information-name=setting-contents  ... [2]

    :
```

Explanation of the contents

- [1] Comment on the runtime initialization file.
- [2] Write the environment variable information common to each program.

Note

- Two or more environment variable information cannot be written per line.
- Do not specify the same environment variable name repeatedly, as operation in this case is not guaranteed.

When the line starts with a semicolon (;), the section from the semicolon up to the line feed character is recognized as a comment. If too many comments lines are included, the processing speed may go down due to skipping comment lines.

No null characters can be specified for an environment variable name. When lines written in incorrect format are detected, the line following it will be analyzed.

Sample coding of runtime initialization file:

```
  ; Environment
CBR_CBRINFO=YES
```

4.1.2.3.2 Searching order of runtime initialization file

The searching order of runtime initialization file is listed below:

1. COBOL.CBR under the directory of the executable program.
2. COBOL.CBR under the directory in which the library executed at the beginning is stored.
3. Runtime initialization file specified by the environment variable CBR_CBRFILE.

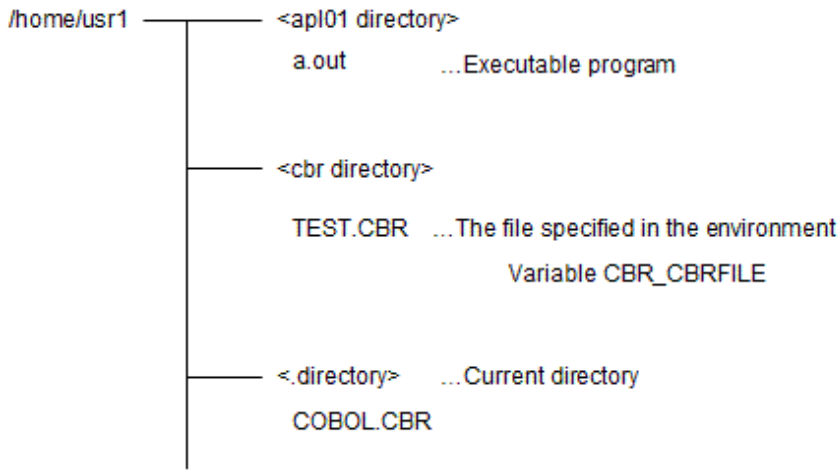
Note

When the executable program contains the COBOL application program name in the library executed at the beginning, the runtime initialization file is searched in the directory of the executable program not searching the directory in which the library executed at the beginning is stored.

Example

Order of searching

The order of searching the runtime initialization file is explained with the following example:



In the above example, the order of searching the runtime initialization is as follows:

1. /home/usr1/apl01/COBOL.CBR
2. /home/usr1/lib01/COBOL.CBR
3. /home/usr1/cbr/TEST.CBR

```
$  PATH=/home/usr1/apl01:$PATH
$  export PATH
$  LD_LIBRARY_PATH=/home/usr1/lib01:$LD_LIBRARY_PATH
$  export LD_LIBRARY_PATH
$  CBR_CBRFILE=/home/usr1/cbr/TEST.CBR
$  export CBR_CBRFILE
$  a.out
```

In the above example, the order of searching the runtime initialization is as follows:

1. /home/usr1/apl01/COBOL.CBR
2. When 1 is not provided : /home/usr1/lib01/COBOL.CBR
3. When 2 is not provided : /home/usr1/cbr/TEST.CBR

COBOL.CBR needs to be stored in the same directory as either the library or the executable. It will be found if it is in the same directory as the executable. If it is not in the same directory as the executable, it will be expected to be in the directory containing the library.

		Directory where library is stored	
		COBOL.CBR is provided	COBOL.CBR is not provided
Directory where executable program is stored	**COBOL.CBR is provided**	Directory where executable program is stored is valid	Directory where executable program is stored is valid
	COBOL.CBR is not provided	Directory where library is stored is valid	-

4.1.2.3.3 Using the runtime initialization file under the directory containing the libraries

When using COBOL applications as components of programs written in other languages, store the COBOL.CBR file in the directory containing the COBOL application libraries. This avoids the problems that arise if COBOL.CBR is stored in the directory of the executable program (which where it typically would be stored if all of the code was written in COBOL). The problems avoided are:

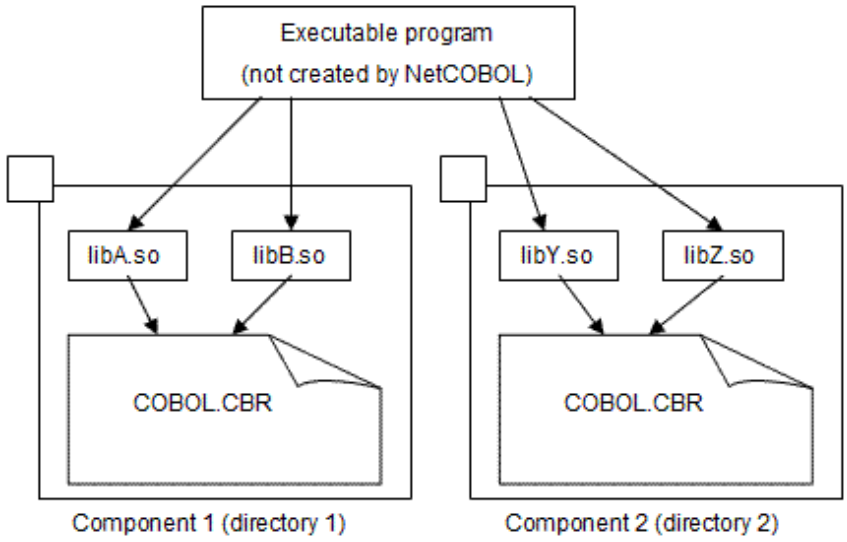
1. COBOL.CBR must be located in the storage directory of the executable program (another product) that starts the COBOL application (library) treated as a component.

   -> The storage directory of the executable program (another product) must be considered.

2. All information on the COBOL application treated as a component must be registered in COBOL.CBR in the storage directory of the executable program.

   -> The COBOL.CBR file size increases, thereby degrading performance.

When the COBOL.CBR is located in the directory containing the COBOL application libraries, the storage location of the executable program written in another language does not need to be considered, as shown in the example below.



For example, in the above figure, applications libA.so and libB.so running in the same runtime environment are located in directory 1, whose COBOL.CBR will remain effective for the execution of that component. Similarly, when the executable program starts component 2, the COBOL.CBR information in directory 2 is effective for the execution of component 2.

Note

- Storing COBOL.CBR in the library directory and not in the executable directory is only supported if the executable is not a COBOL program.
- The COBOL applications (libraries) must be called from the executable program in the dynamic program structure. If they are called in the dynamic link structure, COBOL.CBR in a storage directory of the COBOL applications (libraries) cannot be used.
- COBOL.CBR must never be located in the storage directory of the executable program because a search for the runtime initialization file is made from the storage directory of the executable program.

- Save the COBOL applications (libraries) in one process (runtime environment) to one directory together with COBOL.CBR having information on the process. If the COBOL applications (libraries) in one process (runtime environment) are not saved to one directory, operation becomes unpredictable.
- COBOL.CBR is enabled in one process (runtime environment). Accordingly, if different values are assigned to the same environment variable information in applications, start the respective applications in separate processes.

4.1.2.3.4 How to display runtime initialization file information

Specifying the environment variable CBR_CBRINFO=YES gives you the information on the runtime initialization file. The information is notified as an execution time message when the runtime environment is established.

See "Appendix E Environment Variable List".

4.1.2.4 Setting from the Command Line

This method specifies the content of the runtime environment information as the command argument. In this method, the initialization file (environment variable GOPT) for the runtime parameter of OSIV system form (environment variable MGPRM) and execution can be specified.

See "4.2.3 Specifying the Runtime Initialization File" and "4.2.4 Specifying OSIV system runtime parameters ".

4.1.3 Subprogram entry information

Entry information is required if the executable program is dynamic program structure. However, this can be omitted if the shared object file name of the call program is "libXXX.so" (XXX is a program name). Specify the entry information file name in the "CBR_ENTRYFILE" environment variable. An entry information file is a file that contains an "ENTRY" section that shows the start of the subprogram entry information definition. Entry information is contained in that section.

Entry Information File Format

```
[ENTRY]                    [1]
entry-information
```

Diagram explanation

- [1] This is the section name that shows the start of the subprogram entry information definition

  The section name is fixed as "ENTRY". You can only describe one entry information file in this section.

On a line beginning with a semicolon (;), the part from the semicolon to a line feed character is assumed to be a comment.

If too many comment lines are found, the processing speed may decrease because the comment lines are skipped.

- Since entry information is case sensitive, take care when you specify upper-case letters and lower-case letters.
- Do not specify the same subprogram name repeatedly, as operation in this case is not guaranteed.

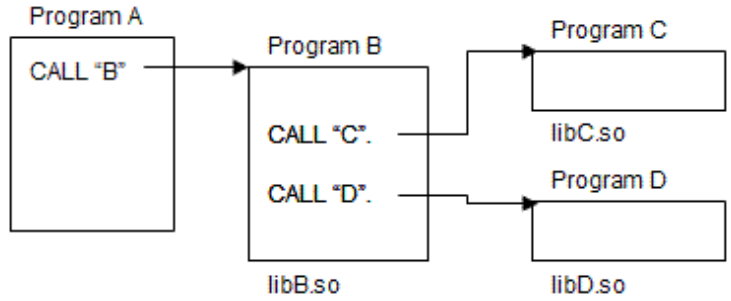
4.1.3.1 Subprogram name format

```
Subprogram-name=shared-object-file-name
```

To relate the subprogram with the shared object file contained in that subprogram, specify the program name of the call program in the subprogram name, and then specify the absolute or relative path of the shared object file name that contains the called program in the shared object file name. If you specify the relative path, this is searched from the directory that has been set in the "LD_LIBRARY_PATH" environment variable. The file extension of the shared object file must be "so".

Example if Shared Objects Have Been Configured In One Subprogram

- Program call relationship



- Entry information file example

```
CBR_ENTRYFILE=FILE
```

- FILE

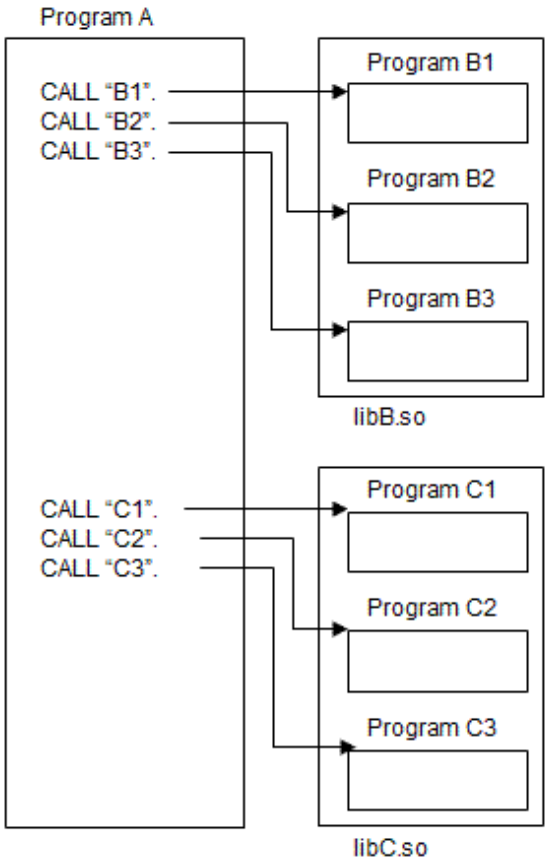
```
[ENTRY]
B=libB.so
C=libC.so
D=libD.so
```

Information

In this example, you can omit the entry information since the shared object file name is "lib*program-name*.so".

Example if Shared Objects Have Been Configured In Several Subprograms

- Program call relationship



- Entry information file example

```
CBR_ENTRYFILE=FILE
```

- FILE

```
[ENTRY]
B1=libB.so
B2=libB.so
B3=libB.so
C1=libC.so
C2=libC.so
C3=libC.so
```

Information

When the CANCEL statement is executed for a called subprogram in a shared object, the shared object is deleted from the memory.

In the above example, when the CANCEL statement is executed for all subprograms in libB.so (B1, B2, and B3), libB.so is deleted from the memory, and when the CANCEL statement is executed for all subprograms in libC.so (C1, C2, and C3), libC.so is deleted from the memory.

If you specify a secondary entry point, or you have to call several subprograms in the same shared object, add "secondary entry point name format" to the subprogram name format.

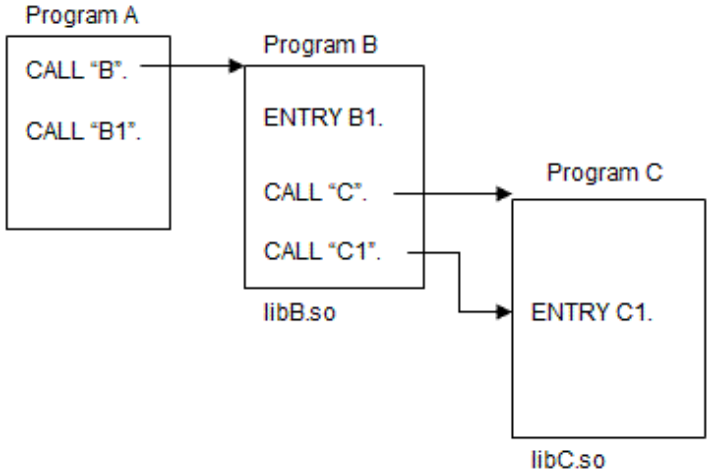
4.1.3.2 Secondary entry point name format

```
Secondary-entry-point-name=subprogram-name
```

In the secondary entry point name, specify the name that is described in the call program ENTRY statement. In the subprogram name, specify the program that contains that ENTRY statement.

Example

- Program call relationship



- Entry information file example

```
CBR_ENTRYFILE=FILE
```

- FILE

```
[ENTRY]
B=libB.so
C=libC.so
B1=B
C1=C
```

4.2 Execution Procedure

This section explains how to execute a COBOL program.

4.2.1 Execution Format of Programs

Executable programs compiled and linked are executed using the name of the file containing the program (executable file) as a command name. An argument can be specified in the command. COBOL program can obtain the values specified in the arguments using the command line argument operation function. See "9.2 Fetching Command Line Arguments" for the command line argument operation function.

The following shows how to execute a COBOL program. If the executable file is not in the current directory during execution of the program, specify the executable file name with an absolute path name.

```
$  file-name  [argument] [-CBL runtime-option] [-CBR runtime-initialization-file-name]
```

Note

-CBR and -CBL can be placed in any order.

How to Specify an Argument

Delimit the arguments with blank spaces. If blank spaces are included in an argument, enclose the argument with double quotation marks (" ").

 Example

Example 1

```
$  PROG1 A B C,D
```

In the above example, three arguments are specified: "A", "B" and "C, D".

Example 2

```
$  PROG1 "A B C,D"
```

In the above example, one argument has been specified: "A B C, D".

Runtime parameters for OSIV system

The argument specified just after the command name is considered to be the runtime parameter of the OSIV system form. Refer to "4.2.4 Specifying OSIV system runtime parameters " for details.

Runtime Options

Runtime options are specified following identifier -CBL. For the format of the runtime options, refer to "4.2.2 Specifying Runtime Options".

 Example

```
$  PROG1 -CBL r20 c20
```

In the above example, r20 and c20 are specified as runtime options.

Specifying an Initialization File Name

Specify an initialization file following identifier -CBR. For the format of the runtime initialization file name, refer to "4.2.3 Specifying the Runtime Initialization File".

 Example

```
$  PROG1 -CBR abc.ini
```

In the above example, the file abc.ini in the current directory is specified as the runtime initialization file name.

4.2.2 Specifying 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 as arguments in the command with the following format:

```
-CBL runtime-options
```

You can specify them in the environment variable GOPT.

```
$  GOPT=runtime-options;export GOPT
```

Types and the specification formats of the runtime option are listed in "Table 4.1 Runtime Options". Use a comma (,) to delimit multiple runtime options.

Table 4.1 Runtime Options

Function	Option
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]
Memory capacity used by PowerBSORT	[smsize valuek]

Information

If the runtime option specified by argument overlaps with GOPT, the argument is given priority over GOPT.

4.2.2.1 [r count | nor] (Set the trace data limit, and suppress the TRACE function)

Set the r count option to change the amount of trace information produced by the TRACE function. Specify the trace data limit from 1 to 999999.

Use 'nor' to suppress the TRACE function.

This option is effective in programs defined with the -Dr option or the TRACE compiler option during compilation.

See "3.3.1.4 -Dr (Specification for using TRACE function)" and "A.2.50 TRACE (whether the TRACE function should be used)".

4.2.2.2 [c count | { noc | nocb | noci | nocn | nocp }] (Set the number of CHECK messages, and suppress the CHECK function)

Set c count option to change the process count when an error is detected by the CHECK function. Specify the process count in the range from 0 to 999999. A value of 0 indicates 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 in programs defined with the -Dk option, the compiler option CHECK(ALL) or the corresponding compiler option during compilation.

See "3.3.1.3 -Dk (Specification for using CHECK function)" and "A.2.5 CHECK (whether the CHECK function should be used)".

4.2.2.3 [svalue] (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. The values can be 0 or 1. If omitted, "S00000000" is assumed. SWITCH-8 is equivalent to SWITCH-O. When SWITCH-8 is used, therefore, the switch values correspond to SWITCH-8, SWITCH-1 to SWITCH-7 from the left.

4.2.2.4 [smsizevaluek] (Memory capacity used by PowerBSORT)

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. The specified value is set for the memory_size of the BSRTPRIM structure of PowerBSORT. For details about valid values, refer to the PowerBSORT "Online Manual".

Although this option is equivalent to the smsize runtime option 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 runtime option has the next highest priority and the SMSIZE() compilation option has the third highest priority.

4.2.3 Specifying the Runtime Initialization File

You can specify the runtime initialization file for a COBOL program at runtime. The runtime initialization file can be specified as an argument in the command with the following format:

```
-CBR runtime-initialization-file-path-name
```

To specify the runtime initialization file, provide an initialization file following identifier -CBR. If the file name includes spaces, enclose it with double quotation marks (" ").

Example

```
a.out -CBR abc.init
```

In the above example, the file abc.init is specified as the runtime initialization file.

4.2.4 Specifying OSIV system runtime parameters

OSIV is the generic name for a system that operates via a global server or PRIMEFORCE such as OSIV/MSP and OSIV/XSP.

Passing a parameter using an OSIV system

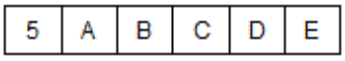
- COBOL program

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    A.
*>   :
DATA DIVISION.
LINKAGE SECTION.
 01 PRM.
  03 PRM-LENGTH  PIC 9(4) BINARY.
  03 PRM-STRINGS.
   05 CHR PIC X
          OCCURS 1 TO 100 TIMES DEPENDING ON PRM-LENGTH.
PROCEDURE DIVISION USING PRM.
```

- Command

```
CALL 'X9999.A.LOAD(A)''ABCDE'
```

- Parameter



Passing the parameter in the system

- The runtime parameter is specified just after the command name

```
$ PROG1 "ABCDE"
```

- The runtime parameter is specified for environment variable MGPRM

```
$ MGPRM="ABCDE"; export MGPRM
```

Note

- Only one OSIV system runtime parameter can be passed.
- The maximum length of the parameter character string is 100 bytes.
- The effective length of the parameter character string is up to the parameter length that is set.
- The area beyond the parameter length cannot be referenced.
- The value cannot be set to the area of the parameter character string.
- If the runtime parameter specified by argument overlaps with @MGPRM, the argument is given priority over @MGPRM.

4.3 Specifying How to Output Execution Time Messages

The following table lists output destinations of execution time messages output by the COBOL runtime system. It also lists environment variables specifying severity levels for the output of messages.

Output destination	Output destination specification	Severity code specification
System standard error output (stderr)	CBR_MESSOUTFILE (See "4.3.2 Outputting Error Messages to Files")	CBR_MESS_LEVEL_CONSOLE (See "4.3.1 Specifying execution time message severity levels")
Interstage Business Application Server general log	CBR_COMPOSER_MESS (See "4.3.4 Outputting execution time messages to the Interstage Business Application Server general log")	--
Syslog	--	CBR_MESS_LEVEL_SYSLOG (See "4.3.3 Outputting execution time messages to Syslog")

4.3.1 Specifying execution time message severity levels

The CBR_MESS_LEVEL_CONSOLE environment variable can be specified to suppress the output of execution time messages by the runtime system, or to specify severity codes for the output of execution time messages.

```
CBR_MESS_LEVEL_CONSOLE=[severity-code]
```

Example

```
$ CBR_MESS_LEVEL_CONSOLE=I; export CBR_MESS_LEVEL_CONSOLE
```

Execution time messages with severity codes of I or higher are output.

The following parameters can be specified for CBR_MESS_LEVEL_CONSOLE:

- NO: No execution time message is output.
- I: Messages with severity codes of I or higher are output.
- W: Messages with severity codes of W or higher are output.
- E: Messages with severity codes of E or higher are output.
- U: Messages with severity codes of U are output.

Note

- If CBR_MESS_LEVEL_CONSOLE is not specified, or if a parameter other than the above parameters is specified on the right side, messages with severity codes of I or higher are output (same as the I specification).
- CBR_MESS_LEVEL_CONSOLE specification is also effective for execution time messages that are output to the file specified by the CBR_MESSOUTFILE environment variable.
- If NO is specified for CBR_MESS_LEVEL_CONSOLE, no execution time message is output. This makes it difficult to determine the cause of an execution time error. Only specify NO after understanding the meaning of the error completely.

4.3.2 Outputting Error Messages to Files

Specifying the environment variable CBR_MESSOUTFILE records runtime messages produced by the runtime system and by the result of DISPLAY statements on SYSERR to a file.

You can use this function to log error messages that occur in a program running under several application servers, including Interstage and Netscape Application Server.

Error messages that occur in a program running under several application servers are output to the standard error output of the server. The standard error output is different according to the server. Specify the environment variable CBR_MESSOUTFILE to ensure that you get the runtime messages.

```
CBR_MESSOUTFILE=message-output-file-name
```

Example

```
$  CBR_MESSOUTFILE=errmsg.log; export CBR_MESSOUTFILE
```

The file errmsg.log is specified as an error message output file.

Note

- An absolute or relative path is available for a file name. When a relative path is specified, it will be the path from the current directory.
- If there is another file with the same name, the information is appended to the file.
- If you specify the file specified as an output file of the input/output function, the content of the file output is not guaranteed.
- The maximum file size is 2GB.
- When the messages cannot be output to the file, they are output to the standard error output.
- The character code of output file is UTF-8.

4.3.3 Outputting execution time messages to Syslog

The CBR_MESS_LEVEL_SYSLOG environment variable can be specified to suppress output of execution time messages to Syslog by the runtime system or to change severity codes for output of execution time messages.

```
CBR_MESS_LEVEL_SYSLOG=[severity-code]
```

Example

```
$  CBR_MESS_LEVEL_SYSLOG=I; export CBR_MESS_LEVEL_SYSLOG
```

Execution time messages with severity codes of I or higher are output to Syslog.

The following parameters can be specified for CBR_MESS_LEVEL_SYSLOG:

- NO: No execution time message is output to Syslog.
- I: Messages with severity codes of I or higher are output to Syslog.
- W: Messages with severity codes of W or higher are output to Syslog.
- E: Messages with severity codes of E or higher are output to Syslog.
- U: Messages with severity codes of U are output to Syslog.

Note

If CBR_MESS_LEVEL_SYSLOG is not specified, or if a parameter other than the above parameters is specified, messages with severity codes of U are output to Syslog (same as the U specification).

Information

LOG_USER is specified as a Syslog parameter facility for messages output to Syslog. The Syslog parameter level varies depending on the severity code:

- I: LOG_INFO
- W: LOG_WARNING
- E: LOG_ERR
- U: LOG_ALERT

Syslog operations can suppress output and can change destinations according to the parameter facility and parameter level. If messages are not output as specified by CBR_MESS_LEVEL_SYSLOG, check Syslog settings (syslog.conf).

4.3.4 Outputting execution time messages to the Interstage Business Application Server general log

The CBR_COMPOSER_MESS environment variable can be specified so that execution time messages output by the runtime system are output to the Interstage Business Application Server general log.

```
CBR_COMPOSER_MESS=[Management-name-defined-by-log-definition-file]
```

Example

```
$  CBR_COMPOSER_MESS=mylog; export CBR_COMPOSER_MESS
```

Execution time messages are output according to the definition of mylog, which is the management name defined by log definition file.

For details on the management name defined by log definition file, see the Interstage Business Application Server manual.

Note

The specification of each of the following environment variables is disabled for execution time messages that are output to the general log, and such messages of all severity levels are output to the general log:

- CBR_MESS_LEVEL_CONSOLE
- CBR_MESS_LEVEL_SYSLOG

Information

The correspondence between execution time message severity levels and general log output levels is as follows:

- I: Output at level 6
- W: Output at level 4
- E: Output at level 3
- U: Output at level 1

4.4 Termination Status

When you terminate a COBOL program using STOP RUN or an EXIT PROGRAM statement in the main program, the value of the special register PROGRAM-STATUS will be a return value from the COBOL program. The special register PROGRAM-STATUS is an alphanumeric string implicitly declared as PIC S9(18) COMP-5 that can be set in a COBOL program. By referencing this value in the shell script that invoked the COBOL program, a formatted job application including execution control can be easily constructed. The following is an example:

- COBOL source program

```
 IDENTIFICATION DIVISION.
  PROGRAM-ID. PROG1.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
     ARGUMENT-VALUE IS nmemonic-name.
 DATA DIVISION.
  WORKING-STORAGE  SECTION.
  01  argument-value-1  PIC  X.
 PROCEDURE  DIVISION.
     ACCEPT  argument-value-1  FROM  nmemonic-name.
     IF  argument-value-1 >= "0"  AND  argument-value-1 <= "9"
      THEN MOVE  0 TO PROGRAM-STATUS
      ELSE MOVE  1 TO PROGRAM-STATUS
     END-IF.     EXIT  PROGRAM.
 END  PROGRAM  PROG1.
```

  The value that determines if the argument value specified in PROG1 is a numeric value is returned as a return value.

- Shell script for execution (extract)

```
if PROG1 $1
   then
     echo $1 "copes will be printed"
   else
     echo "error Specify a numeric value"
fi
```

  Determine the return value from PROG1 to display a message.

Note

In order to see the return value of the program using the shell script "echo $?" correctly, set a value that can be represented with 1 byte (0 to 255) in PROGRAM-STATUS.

4.5 Cautions

This section explains the cautions for running programs.

4.5.1 If a Stack Overflow Occurs During COBOL Program Execution

One of the causes of a bus error during COBOL program execution may be that a stack area of the process overflowed. If the cause of the problem is a stack overflow, enlarge the stack size.

Information

You cannot directly recognize that a stack area has overflowed.

How to refer the stack size of the process

You can find the stack size of the process with the following appropriate command:

- Bash

```
$  ulimit -s
```

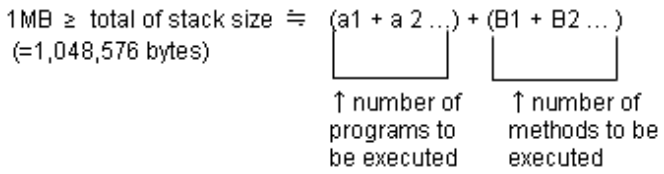
- C shell:

```
%  limit stacksize
```

How to estimate the stack size

The following is a rough value of the stack size required in a COBOL program:

- Formula for calculation



- Description of formula
  - an: Stack size value (n=1, 2, ...used in the program definition
  - Bm: Stack size value (m=1, 2,...) used in the method.

  Each stack size can be obtained from the section size list. For details of the section size list, see "3.1.7.6 Data Area Listings".

How to prevent a stack overflow

To prevent a stack overflow, set a process stack size value larger than the stack size required by the COBOL program used.

Example

Setting the process stack size to 16,384 kilobytes

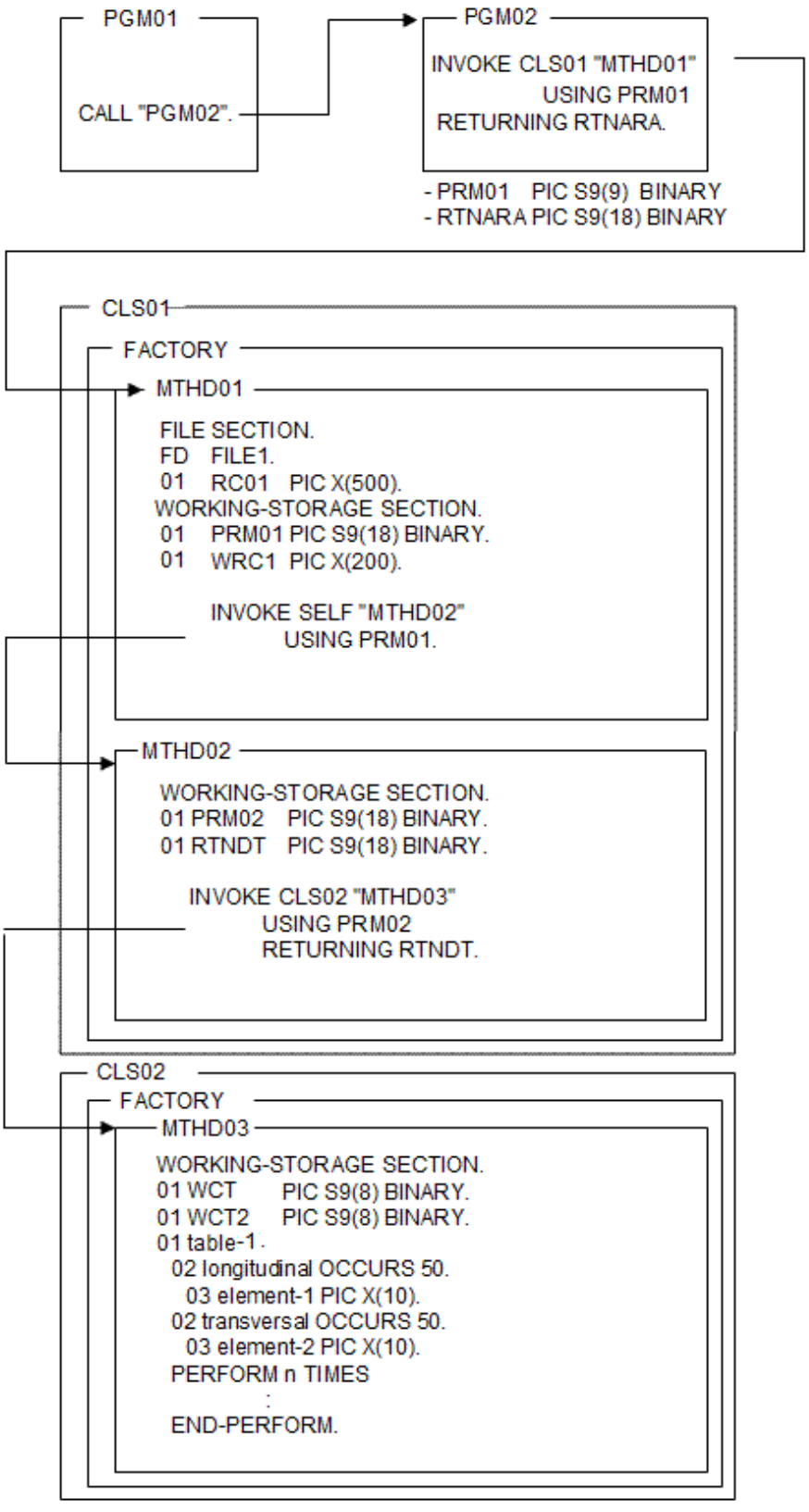
- Bash

```
$ ulimit -s 16384
```

- C shell

```
% limit stacksize 16384
```

Example 1: Non-overflowing case (stack size of 1M bytes)

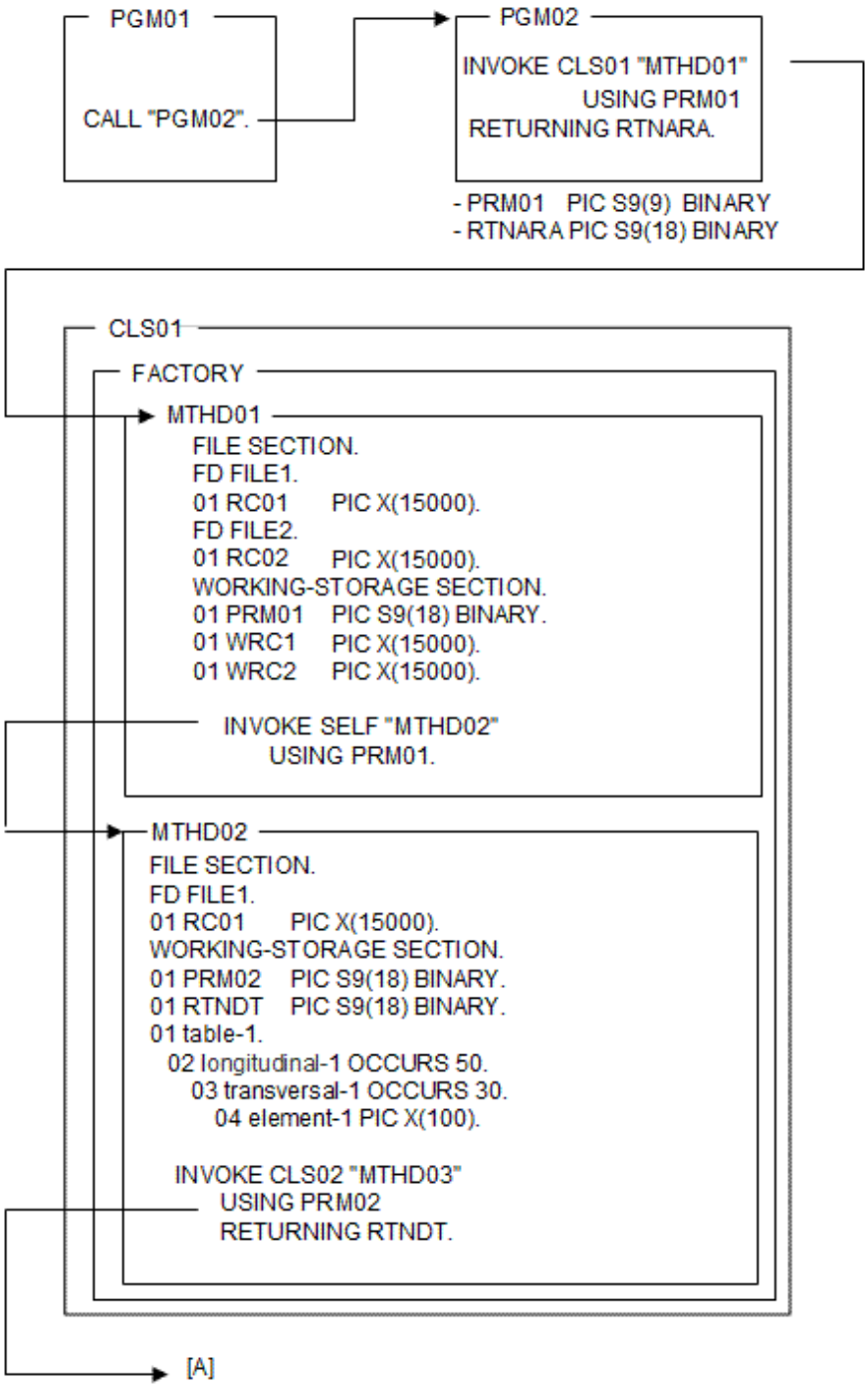


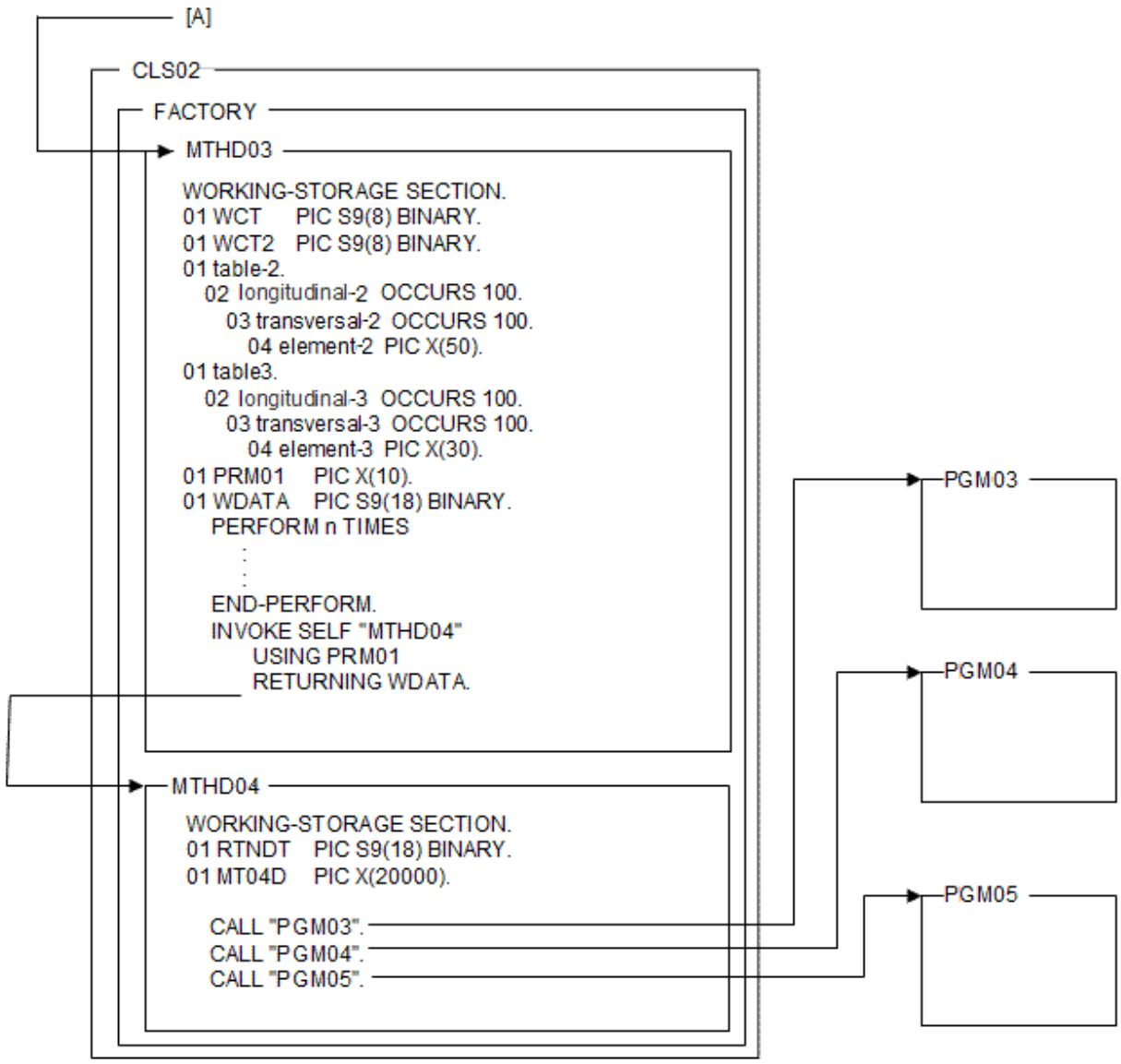
<Stack use status in Example 1>

```
PGM (Program definition)

PGM01                                    1,000 bytes
```

```
PGM02                                   1,000 bytes

CLS (Class definition)

CLS01
METHOD(1)                               2,100 bytes

METHOD(2)                               1,100 bytes

CLS02
METHOD(3)                               2,100 bytes

          Total                         7,300 bytes
```

Example 2: Overflowing case (stack size of 1M bytes)


PGM01
CALL "PGM02".
PGM02
INVOKE CLS01 "MTHD01"
USING PRM01
RETURNING RTNARA.
- PRM01 PIC S9(9) BINARY
- RTNARA PIC S9(18) BINARY
CLS01
FACTORY
MTHD01
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.
MTHD02
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 table-1.
02 longitudinal-1 OCCURS 50.
03 transversal-1 OCCURS 30.
04 element-1 PIC X(100).
INVOKE CLS02 "MTHD03"
USING PRM02
RETURNING RTNDT.
[A]



<Stack use status in Example 2>

```
PGM (program definition)

PGM01                                          1,000 bytes
PGM02                                          1,000 bytes
Maximum stack size of PGM03, PGM04, and PGM05  1,000 bytes

CLS (class definition)

CLS01
METHOD(1)                                     62,000 bytes

METHOD(2)                                    167,000 bytes

CLS02
METHOD(3)                                    802,000 bytes

METHOD(4)                                     22,000 bytes

         Total                             1,056,000 bytes
```

4.5.2 If Virtual Memory Shortages Occur During COBOL Program Execution

Virtual memory shortages when COBOL programs are executing may be caused by one of the causes listed below. 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 if needed.
- 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.
- Too many data items or file declarations use the EXTERNAL phrase in the execution unit.
- Too many objects (instances) are used simultaneously in the execution unit.

Other

- An executed application damaged a memory area.
  -> Search for the cause of the area damage by using a debugger, the CHECK function, or the memory check function, and correct the program.

4.5.3 Fonts Used in the English Environment

Note that the width of a literal may be distorted in an English-language environment when using a proportional font.

Chapter 5 Debugging Programs

This chapter describes a variety of techniques for debugging a COBOL program. These techniques include:

a. Modify the source program directly while you debug it. For example, directly confirm the flow of execution control or data values by adding the debugging logic into the source with a debugging line or inserting the DISPLAY statement to output the data or character stings being executed. For the details on the debugging line, see the "NetCOBOL Language Reference".

b. Specify compiler options (TRACE, CHECK, COUNT) and output the debugging object to detect the logical errors of the program at runtime.

c. Interactively debug the program with the system standard debugger gdb using the object actually being executed.

d. Remote debugging using NetCOBOL Studio included in the NetCOBOL series products that operate under 32-bit Windows.

Method a. can be covered within the COBOL language specification.

A specific process is provided for the methods b. This chapter explains the debugging function and its use method in b.

For details about method c., refer to "18.2 How to Use gdb Command".

For details about method d., refer to "18.1 How to Use Remote Debug Function of NetCOBOL Studio" and "NetCOBOL Studio User's Guide" included in the NetCOBOL series products that operate under Windows.

5.1 Types of Debugging Functions

There are three debugging functions to detect errors in the program:

- Tracing the COBOL statements executed (TRACE function).
- Checking the areas that should not be referred to, data exception, and parameter (CHECK function).
- Outputting the number of executions per COBOL statement or type of statement executed and its ratio (COUNT function).

To use the debugging function, specify compiler options for each debugging function when compiling the program, and set the environment to enable the debugging function when executing the program.

The relationship between the debugging functions and compiler options to specify is listed below in Table below.

Table 5.1 Relation of the debugging functions and compiler options

Function	Overview		Compiler option
	Usage	Processing	
TRACE function	- Determine which statement the program terminated abnormally. - Determine how the program was running before it terminated abnormally. - Confirm the messages output during execution of the program	Outputs the following information: - Trace result of the statements executed. - Line and verb numbers of the statement where the program terminated abnormally. - Name and attribute information of the program that contains the statements executed. - Messages output during execution of the program.	TRACE
CHECK function	- To prevent an operation error of the program due to a memory reference error - To prevent erroneous numbers from causing a program to behave unexpectedly	The following items are checked. Whether the subscript or the index addresses an area outside of the range of a table when that table is referenced - Whether the reference modified exceeds the data length at reference modification	CHECK

Function	Overview		Compiler option
	Usage	Processing	
	- To prevent invalid parameters from causing a program to behave unexpectedly	- Whether the contents of the object word are correct when the data containing the OCCURS DEPENDING ON clause is referenced - Whether the numeric item contains a value of the type that is specified by the attribute. - Whether the divisor in division is not zero. - Whether, in an invoked method, the number of parameters and attributes for a calling method match those for a called method. - Whether the number of parameters and the length of each parameter - when a program is called are the same between the calling program and called program.	
COUNT function	- Determine all routes the program ran through and how it was run - Want to improve the efficiency of the program	Outputs the following information: - The number of executions for each statement in the program and execution ratio of each statement for the total executions in all statements. - The number of executions by the type of the statement in the program and execution ratio by the type of the statement for the total executions in all statements.	COUNT

Note

TRACE and COUNT functions cannot be used at the same time.

5.1.1 Statement number

When you find "Statement number" hereafter, it means:

```
line-number
```

Line number

If the compiler option NUMBER is effective, it takes the format of "[COPY-qualification-value-]user-line-number", and if NONUMBER is enabled, the value will be generated by the compiler from 1 in ascending order by increments of 1.

See "3.1.7.4 Source Program Listing".

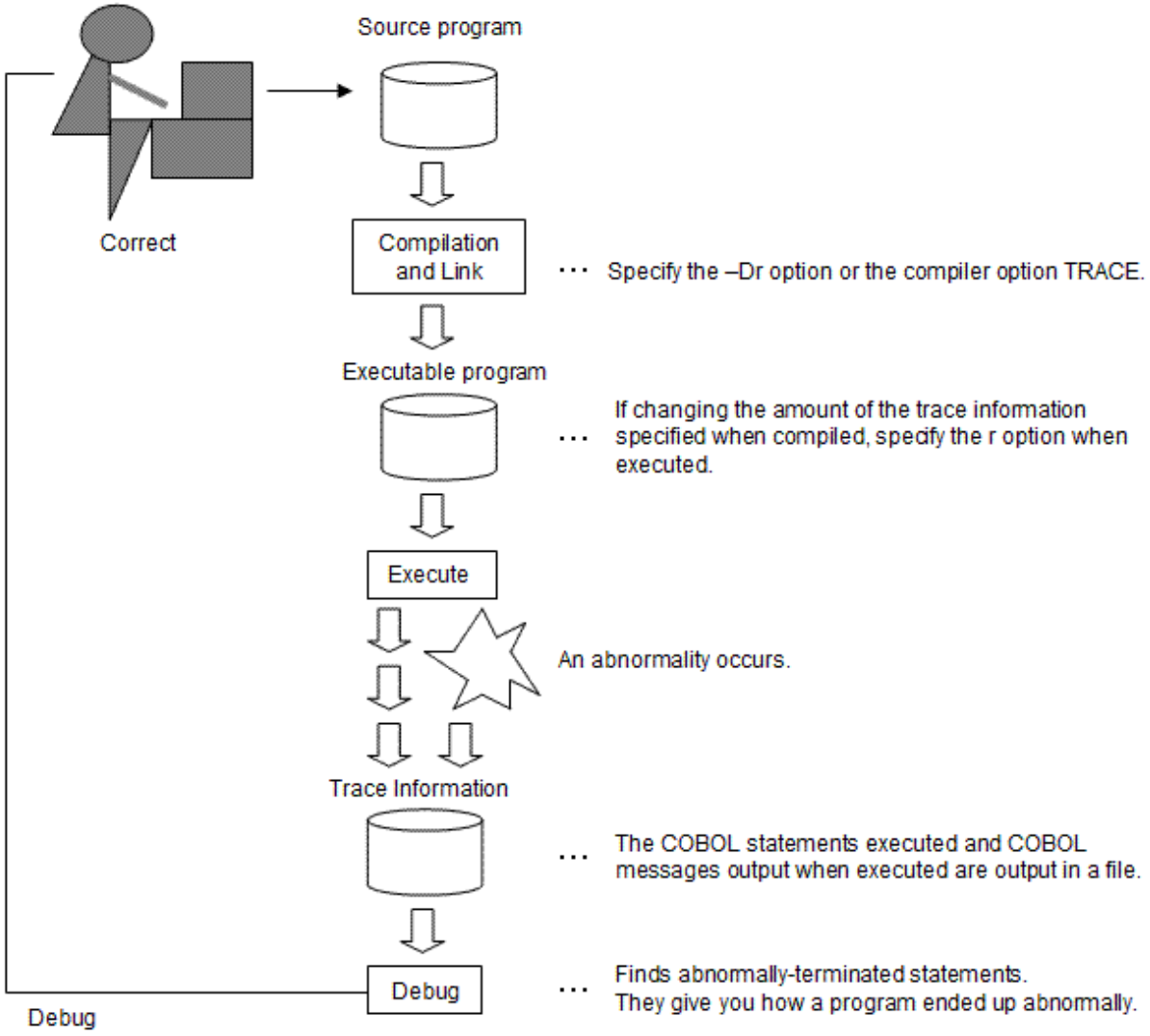
5.2 Using the TRACE Function

TRACE function outputs the trace information collected from the COBOL statements executed immediately before the program terminated abnormally into a file. The trace information gives you the information on what the abnormally-terminated statements are and how they ended up with that way for your debugging. This section explains how to use the TRACE function.

5.2.1 The Flow of Debugging

The following is the flow of debugging using the TRACE function:

Figure 5.1 Debugging using the TRACE function



5.2.2 Trace Information

With the TRACE function, the numbers of the COBOL statements executed immediately before the program terminated abnormally are output as trace information.

Amount of trace information

When the -Dr option, or the compiler option TRACE is specified without the amount of trace information specified, 200 pieces of trace information will be produced. When the compiler option TRACE is specified with the amount specified, the specified amount of trace information will be produced. If there is more than one COBOL program with the TRACE option specified in the execution unit, the amount specified in the compiler option of the program executed first is valid.

You can change the amount of the trace information with the runtime option r at runtime.

The TRACE function can be suppressed by using the option 'nor' at runtime.

Location in which the trace information is stored

Trace information is stored in a file with the file name consisting of the executable program's file name plus the file extension trc. The following is an example:

- Name of the file containing the executable program:

```
/home/xx/PROG1
```

- Name of the file containing the trace information (latest):

```
/home/xx/PROG1.trc
```

Trace information is stored in a file with the extension trc, and the content of the file is moved to a file with extension tro when the amount of stored information reaches the amount specified at compilation or runtime.

To change the name and location of the trace information file, specify the following environment variable at runtime:

```
    CBR_TRACE_FILE=file-name
```

The trace information will be output to a file with the name of the file name specified in the environment variable CBR_TRACE_FILE with extension trc.

Output format of trace information

The following is an output format of the trace information:

```
NetCOBOL DEBUG INFORMATION                          DATE 2010-01-06  TIME 10:10:32
PID=00000123 [1]

TRACE INFORMATION
[2]  [3]                 [4]                    [5]          [6]
1  external-program-name (internal-program-name) compiled on: TID=00000099
2       [7]1100.1 TID=00000099
3          1200.1 TID=00000099
4          1300.1 TID=00000099
5     [8]  1300.2 [9]          [5]
6  class-name [method-name] compiled on:
7          2100.1 TID=00000099
8          2200.1 TID=00000099
9    JMPnnnnI-x xxxxxxxxx xx xxxxxxxxx.  [10]
```

Explanation

- [1] Process ID (hexadecimal 8-digit)
  The number that identifies the process allocated by the operating system when the program is executed.
- [2] Serial number of trace information (decimal 10-digit)
  Displays the number counted every time the trace information is output. The number lets you know what number the information is from the start of the program as the trace information is overwritten into two files one after the other.
- [3] External program name
  The name of an external program.
- [4] Internal program name
  Displayed when an internal program is in operation. Nothing is displayed if the external program is activated.
- [5] Compiled on
  When the external program is activated, the date the running program was compiled is displayed.
- [6] Thread ID (hexadecimal 8-digit)
  The number that identifies the thread allocated by the operating system when the program is executed.
- [7] Statement executed, procedure-name/paragraph-name
  Outputs the statement number of the statement executed, procedure, or paragraph name.
- [8] Class name
  Outputs the class name of the inheritance with the method procedure defined when executing an inherited method.
- [9] Method name
  Outputs the method name.

- [10] Runtime message
  Outputs the message output from the runtime system during execution of the program. See "Runtime Messages" in Appendix N for details.

Trace information file

Trace information is output for each process. In order to prevent overwriting the results of each trace, change the name of the trace output file each time you execute the same program.

Use environment variable CBR_TRACE_PROCESS_MODE to change the trace information file name of each process.

```
CBR_TRACE_PROCESS_MODE=MULTI
```

An example of the use of CBR_TRACE_PROCESS_MODE to change the file name is shown below.

Example

When the environment variable CBR_TRACE_PROCESS_MODE is specified:

```
Executable file name:sample.out
Process-ID:00000EC4
Execution date:12/1/2010
Execution time 10:48:50
```

The name of the new trace information file is:

```
sample-00000EC4_20100112_104850.trc
```

The name of the old trace information file is:

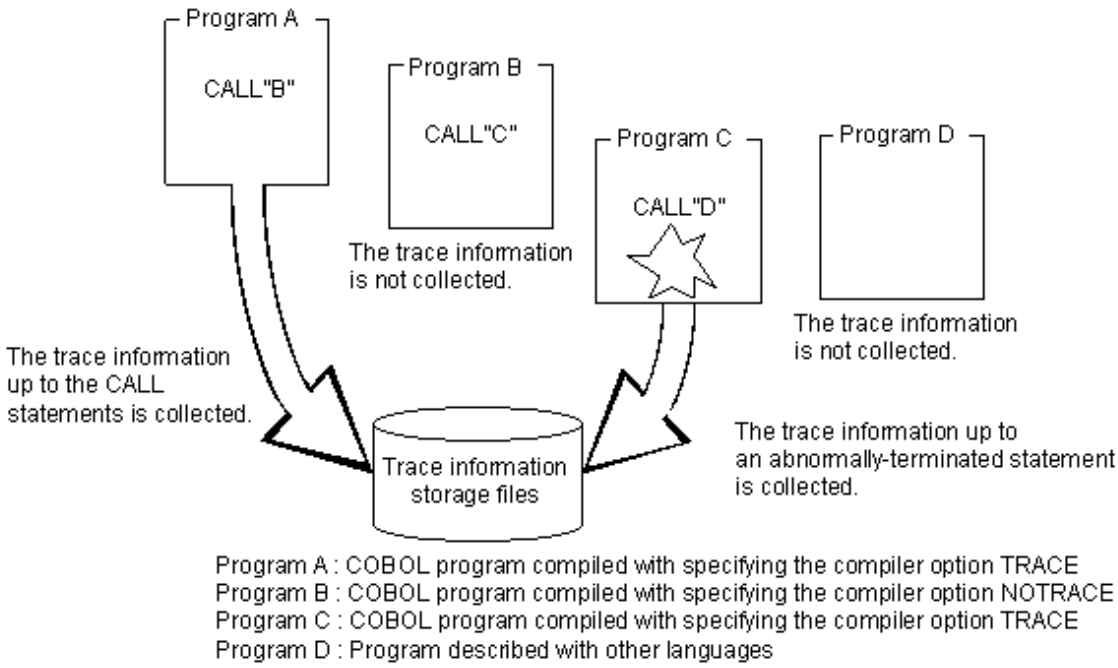
```
sample-00000EC4_20100112_104850.tro
```

5.2.3 Cautions

This section explains the cautions when using the TRACE function.

- Trace information can be collected using the TRACE function only from a COBOL program with the -Dr option or the compiler option TRACE specified.

Figure 5.2 TRACE function information storage



- The TRACE function performs additional processing other than the source code information. Therefore, the size of the program increases, which may degrade the execution speed when using the TRACE function. Use the TRACE function only for debugging a program. After debugging, specify the compiler option NOTRACE to recompile the program.
- You cannot specify zero (0) for the amount of trace information.
- If the trace information file is present and the program is run again, the original content of the file is lost.
- Remove the trace information file when it is no longer needed.
- There is no information that can help you to know it is a prototype-declarative method in the trace information file. When referencing the statement number of a method, refer to the class and method names to determine if the method has been "separated" by the prototype declaration. In the case of a "separated method", the statement number is represented by the line number of the source file of the "separated method". That is not the line number of the source file in the class definition.
- A trace information file is output for each execution file process.
  Information cannot be output from two or more processes to the same file at the same time- this will produce an output error at execution time.
  In order to prevent overwriting the results of each run with trace, change the name of the trace output file each time you execute the same program.

5.3 Using the CHECK Function

The CHECK function prevents the program from working incorrectly. It outputs error messages, lets the program terminate abnormally if an abnormality is detected, and checks the following:

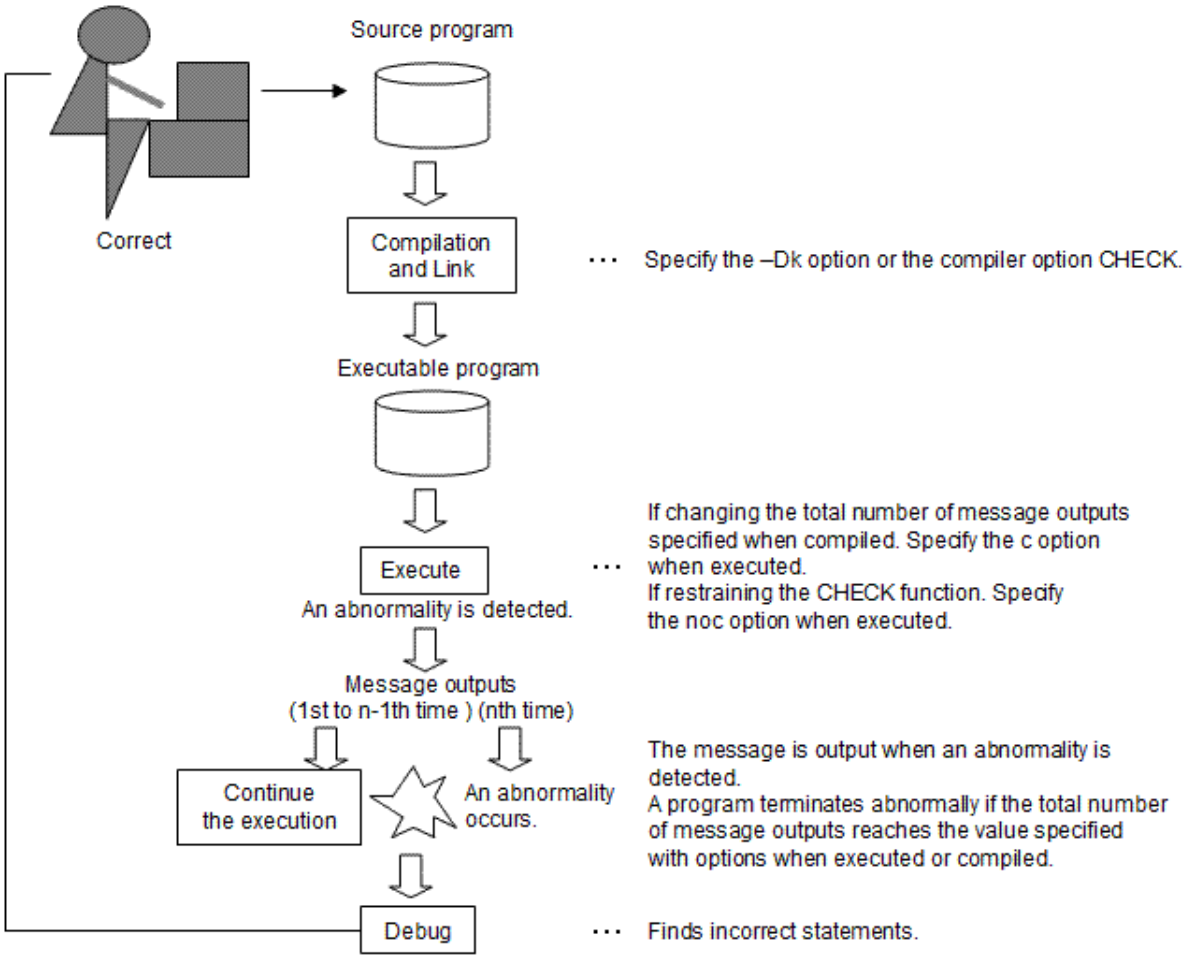
- Subscript/index, reference modification.
- Numeric data exception and divisor zero.
- Parameters in method invocation.
- Internal program call parameters.
- External program call parameters.

This section explains how to use the CHECK function.

5.3.1 The Flow of Debugging

The following is the flow of debugging using the CHECK function:

Figure 5.3 Debugging Flow using CHECK function



5.3.2 Output Messages

The CHECK function outputs an error message if an abnormality is detected after checking. The message is output to where ordinary runtime message are output (standard error output).

The severity code of the messages output with the CHECK function is usually E-level. However, if the total number of message outputs reaches the specified number, it will be U-level. For the severity codes and content of the messages, refer to "NetCOBOL Messages".

The CHECK(PRM) internal program call parameter is output as a diagnostic message at the time of compilation during investigation. It does not mean that it checks the number of times for output.

The CHECK function messages are explained below.

Internal program call parameter investigation

JMN3333I-S

The number of parameters specified in the USING phrase of the CALL statement must be the same as the number of parameters specified in the USING phrase of the PROCEDURE DIVISION.

- Corrective action

  Correct the program so that the number of parameters is the same.

JMN3334I-S

The type of parameter @2@ specified in the USING phrase or the RETURNING phrase of the CALL statement must be the same as the type of parameter @3@ specified in the PROCEDURE DIVISION USING phrase or the RETURNING phrase of program @1@.

- Parameter

  @1@: Program name

  @2@: Identifier

  @3@: Identifier

- Corrective action

  Correct the program so that the class name, FACTORY, and ONLY that are specified in the USAGE OBJECT REFERENCE clause of the object specified in the parameter are the same.

JMN3335I-S

The length of parameter @2@ specified in the USING phrase or RETURNING phrase of the CALL statement must be the same as the length of parameter @3@ specified in the PROCEDURE DIVISION USING phrase or RETURNING phrase of program @1@.

- Parameter

  @1@: Program name

  @2@: Identifier

  @3@: Identifier

- Corrective action

  Correct the program so that the parameter lengths are the same.

JMN3414I-S

A 'RETURNING' phrase must be specified for a CALL statement which calls @1@. There is a RETURNING phrase in the PROCEDURE DIVISION of program @1@.

- Parameter

  @1@: Program name

- Corrective action

  Correct the program so that the existence or non-existence of RETURNING phrase is matched.

JMN3508I-S

A 'RETURNING' phrase cannot be specified for a CALL statement which calls @1@. There is no RETURNING phrase in the PROCEDURE DIVISION of program @1@.

- Parameter

  @1@: Program name

- Corrective action

  Correct the program so that the existence or non-existence of RETURNING phrase is matched.

External program call parameter investigation

JMP0812I-E/U

[PID:xxxxxxxx TID:xxxxxxxx] FAILURE IN '$1' OF CALL STATEMENT. $2. $3.

- Parameter

  $1: Character string indicating the detected error

$2: A program name, or a class name and method name

$3: The statement number

- Corrective action

  Take action as described in "Table: JMP0812I-E/U $1 characters" according to the character string indicated by $1.

Table 5.2 JMP0812I-E/U $1 characters

$1	Action
USING PARAMETER NUMBER	Make the value match the number of parameters specified for USING.
USING nTH PARAMETER (nTH = 1ST, 2ND, 3RD, 4TH...)	Make the value match the size of the n-th parameter specified for USING.
RETURNING PARAMETER	Make the value match the size of the parameter specified for RETURNING.

Total number of message outputs

Specify the total number of messages that can be output in compiler option CHECK at compilation. If more than one COBOL program in the execution unit has the CHECK option specified, the number of outputs specified in the compiler option of the program executed first will take precedence.

If the -Dk option is specified at compilation, 1 is used. You can change the number of messages produced by specifying option c at runtime.

Specifying the runtime option will suppress the CHECK function. Each CHECK function can be suppressed by the following runtime options:

- noc : All of the CHECK function
- nocb : CHECK(BOUND)
- noci : CHECK(ICONF)
- nocn : CHECK(NUMERIC)
- nocp : CHECK(PRM)

The program continues to run until the number of messages produced reaches the specified number after an abnormality is detected.

5.3.3 Example of CHECK Function

This section explains an example of the CHECK function.

Subscript and index check

Compiler options CHECK (BOUND) or CHECK (ALL)

```
000500  77  subscript   PIC S9(4).
000600  01  table-data.
000700      02  table-1  OCCURS 10 TIMES INDEXED BY  index-1.
000800          03  element-1  PIC 9(5).
      *     :
001000 PROCEDURE DIVISION.
001100     MOVE  15       TO  subscript.
001200     ADD    1       TO  element-1 (subscript).
001300     SET  index-1   TO  0.
001400     SUBTRACT  1  FROM  element-1 (index-1).
      *     :
```

When executing ADD/SUBTRACT statements, the following messages are output:

```
JMP0820I-E/U [PID:XXXXXXXX TID:XXXXXXXX] SUBSCRIPT/INDEX IS OUT OF RANGE. PGM=A. LINE=1200.
OPD=ELEMENT-1 (1)
JMP0820I-E/U [PID:XXXXXXXX TID:XXXXXXXX] SUBSCRIPT/INDEX IS OUT OF RANGE. PGM=A. LINE=1400.
OPD=ELEMENT-1 (1)
```

Reference modification check

Compiler options CHECK (BOUND) or CHECK (ALL)

```
000500  77  data-1          PIC X (12).
000600  77  data-2          PIC X (12).
000700  77  refer-length    PIC 9 (4) BINARY.
      *      :
001000 PROCEDURE DIVISION.
001100     MOVE  10   TO  length to refer to.
001200     MOVE   data-1 (1: refer-length) TO
001300            data-2 (4: refer-length).
      *     :
```

When executing a 1200-line MOVE statement, the following message is output for data-2:

```
JMP0821I-E/U [PID:XXXXXXXX TID:XXXXXXXX] REFERENCE MODIFIER IS OUT OF RANGE. PGM=A. LINE=1200.
OPD=DATA-2.
```

Object word check in the OCCURS DEPENDING ON clause

Compiler options CHECK (BOUND) or CHECK (ALL)

```
000500  77  subscript       PIC S9(4).
000600  77  object-number   PIC S9(4).
000700  01  table-0.
000800      02  table-1  OCCURS 1 TO  10 TIMES
000900                    DEPENDING ON  object-number.
001000          03  element    PIC X(5).
      *     :
       PROCEDURE DIVISION.
001200     MOVE   5       TO  subscript.
001300     MOVE  25       TO  object-number.
001400     MOVE  "ABCDE"  TO  element (subscript).
      *     :
```

When executing a 1400-line MOVE statement, the following message is produced for the number of object words:

```
JMP0822I-E/U [PID:XXXXXXXX TID:XXXXXXXX] ODO OBJECT VALUE IS OUT OF RANGE. PGM=A. LINE=1400.
OPD=ELEMENT. ODO=OBJECT-NUMBER.
```

Numeric data exception check

Compiler options CHECK (NUMERIC) or CHECK (ALL)

```
000500  01  character-data  PIC X(4)   VALUE "ABCD".
000600  01  zoned-decimal REDEFINES character-data   PIC S9(4).
000700  01  numeric-data    PIC S9(4).
      *    :
001400 PROCEDURE DIVISION.
001500     MOVE   zoned-decimal  TO  numeric-data.
      *     :
```

When executing the MOVE statement, the following message is output for the zoned decimal:

```
JMP0828I-E/U [PID:XXXXXXXX TID:XXXXXXXX] INVALID VALUE SPECIFIED. PGM=A. LINE=1500. OPD=ZONED-DECIMAL.
```

Divisor zero check

Compiler options CHECK (NUMERIC) or CHECK (ALL)

```
000600  01  dividend  PIC S9(8)  BINARY  VALUE  1234.
000700  01  divisor   PIC S9(4)  BINARY  VALUE  0.
000800  01  result    PIC S9(4)  BINARY  VALUE  0.
```

```
      *     :
001400 PROCEDURE DIVISION.
001500     COMPUTE  result =  dividend /  divisor.
      *     :
```

When executing the COMPUTE statement, the following message is produced for the divisor:

```
JMP0829I-E/U [PID:XXXXXXXX TID:XXXXXXXX] DIVIDED BY ZERO. PGM=A. LINE=1500. OPD=DIVISOR
```

Parameter check in method invocation

Program A

Compiler options CHECK (ICONF) or CHECK (ALL)

```
      *     :
000030  01  PRM-01  PIC X(9).
000040  01  OBJ-U   USAGE IS OBJECT REFERENCE.
      *     :
000050 PROCEDURE DIVISION.
000060     SET     OBJ-U  TO  B.
000070     INVOKE  OBJ-U  "C"  USING BY REFERENCE PRM-01.
      *     :
```

Class B/method C

```
000010 CLASS-ID. B.
      *     :
000030 FACTORY.
000040 PROCEDURE DIVISION.
      *     :
000060 METHOD-ID. C.
      *     :
000080  LINKAGE SECTION.
000090  01  PRM-01  PIC 9(9)  PACKED-DECIMAL.
000100  PROCEDURE DIVISION  USING PRM-01.
      *     :
```

When executing the INVOKE statement in Program A, the following message is generated:

```
JMP0810I-E/U [PID:XXXXXXXX TID:XXXXXXXX] FAILURE IN USING PARAMETER OF THE 'C' METHOD. PARAMETER=1
PGM=A LINE=70..
```

Parameter check in internal program invocation

Compile option CHECK(PRM) or CHECK(ALL)

Program A

```
000001 @OPTIONS CHECK(PRM)
000002 PROGRAM-ID. A.
000003 ENVIRONMENT DIVISION.
000004 CONFIGURATION SECTION.
000005 REPOSITORY.
000006     CLASS CLASS1.
000007 DATA DIVISION.
000008 WORKING-STORAGE SECTION.
000009 01 P1 PIC X(20).
000010 01 P2 PIC X(10).
000011 01 P3 USAGE OBJECT REFERENCE CLASS1.
000012 PROCEDURE DIVISION.
000013     CALL "SUB1" USING P1 P2            *> JMN3333I-S
000014     CALL "SUB2"                        *> JMN3414I-S
000015     CALL "SUB1" USING P1 RETURNING P2  *> JMN3508I-S
000016     CALL "SUB1" USING P2               *> JMN3335I-S
```

```
000017     CALL "SUB3" USING P3               *> JMN3334I-S
000018     EXIT PROGRAM.
000019*
000020 PROGRAM-ID. SUB1.
000021 DATA DIVISION.
000022 LINKAGE SECTION.
000023 01 L1 PIC X(20).
000024 PROCEDURE DIVISION USING L1.
000025 END PROGRAM SUB1.
000026*
000027 PROGRAM-ID. SUB2.
000028 DATA DIVISION.
000029 LINKAGE SECTION.
000030 01 RET PIC X(10).
000031 PROCEDURE DIVISION RETURNING RET.
000032 END PROGRAM SUB2.
000033*
000034 PROGRAM-ID. SUB3.
000035 DATA DIVISION.
000036 LINKAGE SECTION.
000037 01 L-OR1 USAGE OBJECT REFERENCE.
000038 PROCEDURE DIVISION USING L-OR1.
000039 END PROGRAM SUB3.
000040 END PROGRAM A.
```

When program A is compiled, the following compilation diagnostic message is output:

```
** DIAGNOSTIC MESSAGE ** (A)
13: JMN3333I-S  The number of parameters specified in the USING phrase of the CALL statement must be
the same as the number of parameters specified in the USING phrase of the PROCEDURE DIVISION.
14: JMN3414I-S  A 'RETURNING' phrase must be specified for a CALL statement which calls 'SUB2'. There
is a RETURNING phrase in the PROCEDURE DIVISION of program 'SUB2'.
15: JMN3508I-S  A 'RETURNING' phrase cannot be specified for a CALL statement which calls 'SUB1'.
There is no RETURNING phrase in the PROCEDURE DIVISION of program 'SUB1'.
16: JMN3335I-S  The length of parameter 'P2' specified in the USING phrase or RETURNING phrase of the
CALL statement must be the same as the length of parameter 'L1' specified in the PROCEDURE DIVISION
USING phrase or RETURNING phrase of program 'SUB1'.
17: JMN3334I-S  The type of parameter 'P3' specified in the USING phrase or the RETURNING phrase of
the CALL statement must be the same as the type of parameter 'L-OR1' specified in the PROCEDURE
DIVISION USING phrase or the RETURNING phrase of program 'SUB3'.
HIGHEST SEVERITY CODE = S
STATISTICS: HIGHEST SEVERITY CODE=S, PROGRAM UNIT=1
```

Parameter check in external program invocation

When there is a parameter delivery error in a program invocation, the program operates incorrectly because an incorrect location is referenced or updated.

When a COBOL program that was compiled with the CHECK(PRM) compile option specified is called from another COBOL program compiled with the CHECK(PRM) compile option specified, a message is output if the number of parameters and the parameter lengths do not match.

```
000010 @OPTIONS CHECK(PRM)
000020 IDENTIFICATION  DIVISION.
000030 PROGRAM-ID.     A.
000040 DATA            DIVISION.
000050 WORKING-STORAGE SECTION.
000060 01  USE-PRM01   PIC 9(04).
000070 01  USE-PRM02   PIC 9(04).
000080 01  RET-PRM01   PIC 9(04).
000090 PROCEDURE       DIVISION.
000100     CALL "B" USING USE-PRM01 USE-PRM02
```

```
000110              RETURNING RET-PRM01.
000120 END PROGRAM     A.
```

```
000000 @OPTIONS CHECK(PRM)
000010 IDENTIFICATION  DIVISION.
000020 PROGRAM-ID.     B.
000030 DATA            DIVISION.
000070 LINKAGE         SECTION.
000080 01  USE-PRM01   PIC 9(08).
000090 01  USE-PRM02   PIC 9(04).
000100 01  RET-PRM01   PIC 9(04).
000120 PROCEDURE       DIVISION USING USE-PRM01 USE-PRM02
000130                          RETURNING RET-PRM01.
000140 END PROGRAM     B.
```

When the CALL statement of program A is executed, the following message is output:

```
JMP0812I-U [PID:0000441F TID:00000001] FAILURE IN 'USING 1ST PARAMETER' OF CALL STATEMENT. PGM=A.
LINE=10.
```

5.3.4 Cautions

This section explains the cautions when using the CHECK function.

- Do not fail to use the CHECK function to check the program to correct the errors detected. If the errors are not corrected, serious problems may occur including memory destruction, etc. If you run an application program without the errors corrected, the behavior of the program is not guaranteed.
- Specifying the number of message outputs lets the program continue running after an abnormality is detected. However, the behavior of the program is not guaranteed in that case.
- The CHECK function performs additional processing other than the source code in the COBOL program, including checking the data content. Therefore, the size of the program increases, which may degrade the execution speed when using the CHECK function. Use the CHECK function only for debugging a program. After the debugging, specify the compiler option NOCHECK to recompile the program.
- In an arithmetic statement with ON SIZE ERROR or NOT ON SIZE ERROR specified, a check for a zero as the divisor is made with ON SIZE ERROR, and no such check is made with CHECK(NUMERIC).
- When the divisor zero check is performed, the program terminates abnormally regardless of the number of messages output.
- If an internal program is called by a CALL statement that has an identifier specified as the program name, CHECK(PRM) makes no check.
- To use CHECK(PRM) to check external program calling parameters, both the calling program and called program must be compiled with the CHECK(PRM) compile option specified. If a program created in another language is called by a CALL statement or is the calling program, the parameters are not checked.
- If the difference between the number of parameters of the calling program and the number of parameters of the called program is four or more when external program calling parameters are checked by CHECK(PRM), no error can be detected.
- The maximum duration is used instead of the execution duration for a parameter of a variable-length item when parameters are checked by CHECK(PRM). Therefore, a message may be output for a variable-length item even when the parameter length actually matches.
- If RETURNING is not specified for the calling program or called program when external program calling parameters are checked by CHECK(PRM), PROGRAM-STATUS is implicitly transferred and an 8-byte RETURNING parameter is assumed specified for the check.
- The CHECK function is effective only in programs that specify the CHECK option. When two or more programs are linked, specify the CHECK option only in the programs that you want to target.

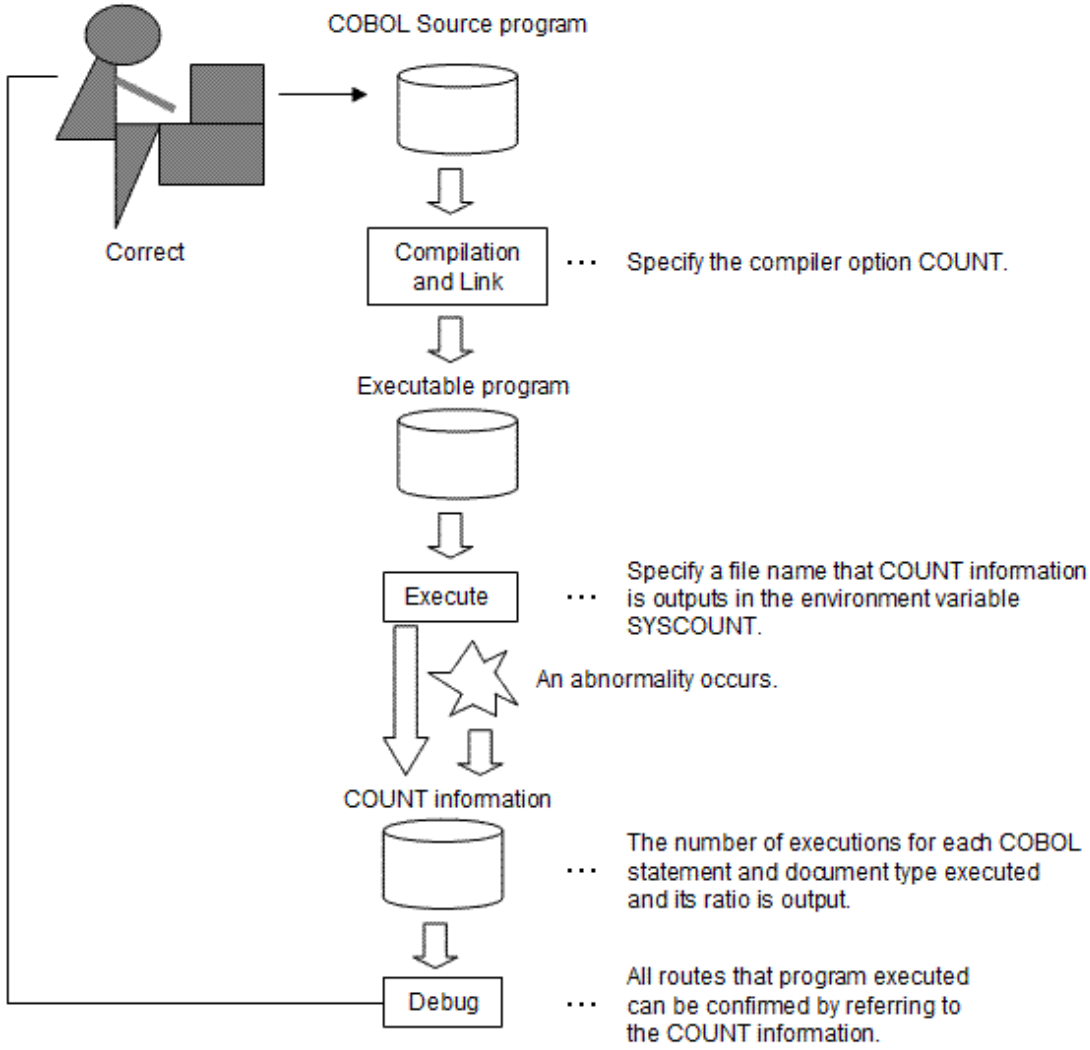
5.4 Using the COUNT Function

The COUNT function displays the number of executions for each statement in the source program and the execution ratio of each statement for the total executions in all statements. It also displays the number of executions by type of statement executed in the source program and its ratio. With the COUNT function, users are able to know the exact frequency of the executions of each statement to optimize the program.

5.4.1 The Flow of Debugging

The following is the flow of debugging using the COUNT function:

Figure 5.4 Debugging Flow using COUNT function



5.4.2 COUNT Information

When the compiler option COUNT is valid, the information is output to the file specified by the environment variable SYSCOUNT. See "A.2.8 COUNT (whether the COUNT function should be used)".

COUNT information save destination

COUNT information is stored in the file specified by the SYSCOUNT environment variable. An example is as follows:

```
SYSCOUNT=file-name[,MOD]
```

- The name of the COUNT information output file used with the COUNT function is specified in file-name.
- An absolute path or relative path can be specified for file-name. If a relative path is specified, it is relative to the current directory.
- To use a file name or directory name containing a comma (,), it must be enclosed in double quotation marks (").
- If an existing file has the name specified in file-name, the file is overwritten. If no existing file has the specified name, a file is created.
- If a character string other than MOD is specified in the second argument, message JMP0726I-W is output. In this case, COUNT information is not output.
- Refer to the COBOL file system in "File system functions" for the maximum file size.

Output format of COUNT information

The following is an output format of the COUNT information:

```
 [1]
NetCOBOL  COUNT INFORMATION (END OF RUN UNIT)        DATE 2010-01-06  TIME 11:02:19
  PID=000014B1 TID=00000001

   [2]
   STATEMENT EXECUTION COUNT    PROGRAM-NAME : COUNT-PROGRAM
       [3]                                          [5]          [6]
       STATEMENT       [4]                          EXECUTION    PERCENTAGE
         NUMBER        PROCEDURE-NAME/VERB-ID         COUNT            (%)
     ------------  ----------------------------  ---------------  --------
               15  PROCEDURE DIVISION   COUNT-PROGRAM
               17      DISPLAY                                  1   14.2857
               19      CALL                                     1   14.2857
               21      DISPLAY                                  1   14.2857
               23      STOP RUN                                 1   14.2857
               31  PROCEDURE DIVISION   INTERNAL-PROGRAM
               33      DISPLAY                                  1   14.2857
               35      INVOKE                                   1   14.2857
               37      EXIT PROGRAM                             1   14.2857
     ------------  ----------------------------  ---------------  --------
                                                                7
   [7]   VERB EXECUTION COUNT         PROGRAM-NAME : COUNT-PROGRAM
                                         [11]                      [13]
     [8]         [9]          [10]       PERCENTAGE [12]           PERCENTAGE
     VERB-ID     ACTIVE VERB  TOTAL VERB       (%)  EXECUTION COUNT      (%)
    ------------ ----------- ------------ --------  --------------- --------
    CALL                   1            1 100.0000                1  25.0000
    DISPLAY                2            2 100.0000                2  50.0000
    STOP RUN               1            1 100.0000                1  25.0000
    ------------ ----------- ------------ --------  --------------- --------
                           4            4 100.0000                4
   [7]   VERB EXECUTION COUNT         PROGRAM-NAME : COUNT-PROGRAM
                                                (INTERNAL-PROGRAM)
                                         [11]                      [13]
     [8]         [9]          [10]       PERCENTAGE [12]           PERCENTAGE
     VERB-ID     ACTIVE VERB  TOTAL VERB       (%)  EXECUTION COUNT      (%)
    ------------ ----------- ------------ --------  --------------- --------
    DISPLAY                1            1 100.0000                1  33.3333
    EXIT PROGRAM           1            1 100.0000                1  33.3333
    INVOKE                 1            1 100.0000                1  33.3333
    ------------ ----------- ------------ --------  --------------- --------
                           3            3 100.0000                3
   [14]   PROGRAM EXECUTION COUNT      PROGRAM-NAME : COUNT-PROGRAM
                                         [18]                      [20]
    [15]         [16]         [17]       PERCENTAGE [19]           PERCENTAGE
    PROGRAM-NAME ACTIVE VERB  TOTAL VERB       (%)  EXECUTION COUNT      (%)
    ------------ ----------- ------------ --------  --------------- --------
    COUNT-PROGRAM          4            4 100.0000                4  57.1429
```

```
    INTERNAL-PROGRAM       3            3 100.0000               3  42.8571
    ------------ ---------- ------------ --------  --------------- --------
                           7            7 100.0000               7
```

```
 [1]
NetCOBOL  COUNT INFORMATION (END OF RUN UNIT)        DATE 2010-01-06  TIME 11:02:19
  PID=000014B1  TID=00000001

   [2]
   STATEMENT EXECUTION COUNT    CLASS-NAME : COUNT-CLASS
       [3]                                          [5]          [6]
       STATEMENT       [4]                          EXECUTION    PERCENTAGE
         NUMBER        PROCEDURE-NAME/VERB-ID         COUNT            (%)
     ------------  ------------------------------  ---------------  --------
               15  PROCEDURE DIVISION   COUNT-METHOD
               16      DISPLAY                                  1   50.0000
               37      EXIT METHOD                              1   50.0000
     ------------  ------------------------------  ---------------  --------
                                                                  2
   [7]   VERB EXECUTION COUNT         CLASS-NAME  : COUNT-CLASS
                                 METHOD-NAME : COUNT-METHOD
                                        [11]                      [13]
     [8]         [9]          [10]      PERCENTAGE [12]           PERCENTAGE
     VERB-ID     ACTIVE VERB  TOTAL VERB       (%)  EXECUTION COUNT      (%)
    ------------ ----------- ------------ --------  --------------- --------
    DISPLAY                1            1 100.0000               1  50.0000
    END METHOD             1            1 100.0000               1  50.0000
    ------------ ----------- ------------ --------  --------------- --------
                           2            2 100.0000               2
   [14]
   METHOD EXECUTION COUNT       CLASS-NAME : COUNT-CLASS
                                        [18]                      [20]
     [15]        [16]         [17]      PERCENTAGE [19]           PERCENTAGE
    METHOD-NAME  ACTIVE VERB  TOTAL VERB       (%)  EXECUTION COUNT      (%)
    ------------ ----------- ------------ --------  --------------- --------
    COUNT-METHOD           2            2 100.0000               2 100.0000
    ------------ ----------- ------------ --------  --------------- --------
                           2            2 100.0000               2
   [21]
   PROGRAM/CLASS/PROTOTYPE METHOD EXECUTION COUNT
    [22]                                 [25]                      [27]
    PROGRAM/CLASS [23]        [24]       PERCENTAGE [26]           PERCENTAGE
    /METHOD-NAME ACTIVE VERB  TOTAL VERB       (%)  EXECUTION COUNT      (%)
    ------------ ----------- ------------ --------  --------------- --------
    COUNT-PROGRAM          7            7 100.0000               7 100.0000
    COUNT-CLASS            2            2 100.0000               2 100.0000
    ------------ ----------- ------------ --------  --------------- --------
                           9            9 100.0000               9
```

Explanation

- [1] Output file of the COUNT function where () indicates when to output the COUNT information. There are four different times information can be produced:

  - **END OF RUN UNITS**

    Information at the end of the execution unit of COBOL (at runtime of the STOP RUN statement, or the EXIT PROGRAM statement of the main program).

  - **ABNORMAL END**

    Information at abnormal termination.

- **END OF INITIAL PROGRAM**

  Information when the program with INITIAL attribute is terminated. However, it is not true of the internal program's termination.

- **CANCEL PROGRAM**

  Information when a program with the compiler option COUNT enabled has been canceled by the CANCEL statement. However, it is not true of the internal program's termination.

- [2] Indicates what is output hereafter; the list of the number of executions in source program image. This information is output by the unit of compilation of the source program. When the compilation takes place by the unit of source program, the external program name is shown in PROGRAM-NAME. When the compilation takes place by the unit of class, the class name is shown in CLASS-NAME. When the compilation takes place by the unit of method, the class name is shown in CLASS-NAME and the method name in METHOD-NAME.

- [3] Displays the statement number in the following format:

```
[COPY-qualification-value -] line-number
```

  If one line contains more than one statement, the line number of the second statement onwards has the same value as the firsts.

- [4] Displays the procedure name and statements. At the beginning of the procedure division, the program or method name follows "PROCEDURE DIVISION".

- [5] Displays the number of executions. The total execution count is at the end.

- [6] Displays the ratio of the statement for the total executions.

- [7] Indicates what is output hereafter; the list of the number of executions per statement. This information is output in the unit of program or method. Therefore, if a class contains programs with internal programs or more than one method, the list of the number of executions is output by statements. The program name is described in the following format in PROGRAM-NAME:

```
PROGRAM-NAME: program-name
          [(called-internal-program-name)]
```

- [8] Outputs the type of statements in alphabetical order. All statements written in the corresponding source programs are output.

- [9] Displays the numbers of commands actually run for each statement in the source program.

- [10] Displays the number of each statement in the source program.

- [11] Displays the execution ratio of each statement in the source program.
  The formula is [9] / [10] * 100.

- [12] Displays the number of executions for each statement. The total execution count is at the end.

- [13] Displays the execution ratio of each statement for the total executions.
  Calculated with the number of executions of each statement / number of executions * 100.

- [14] Indicates what is output hereafter is the list of the number of executions by program or method. The list is output when the program or class has internal programs.

- [15] Outputs the program or method name in the order it appears in the source program.

- [16] Displays the number of statements actually executed for each statement in the source program.

- [17] Displays the number of each statement in the source program.

- [18] Displays the execution ratio of each statement in the source program for the total executions.
  The formula is [16] / [17] * 100.

- [19] Displays the number of executions of the statements in each program or method. Displays the total at the end.

- [20] Displays the ratio of the executed statements in each program or method for the total executions.
  Uses either of the following formulas; number of executions of the statements in each program / total execution count of the statements in all programs * 100, or number of executions of the statements in each method / total execution count of the statements in all methods * 100.

- [21] Indicates what is output hereafter is the list of the number of executions per source program (by the unit of compilation). If there is more than one source program (compilation unit) in the execution unit, this is displayed at the last after the above information is output for the number of programs repeatedly.
- [22] Displays the external program name, class name and method name of the prototype.
- [23] See [16].
- [24] See [17].
- [25] See [18].
- [26] Displays the number of executions for the statements per compilation unit. Displays the total at the end.
- [27] Displays ratio of the executed statements for the total executions per compilation unit.
  Use the formula; number of executions for the statements per compilation unit / total execution count for the statements in all compilation units * 100.

5.4.3 Debugging a Program Using the COUNT Function

The following is an example of debugging a program using the COUNT function:

- Confirm all routes the program ran through.

  The output list of the COUNT function displays the execution count of the statements, thus checking this information enables you to confirm all routes the program ran through.

- Efficiency of the program.

  Checking the ratio of the execution count for each statement in the output list of the CHECK function and the ratio of the execution count for the statements per program search portions that are frequently accessed in the program. Optimizing the statements in these portions improves the efficiency of the whole program.

5.4.4 Cautions

This section explains the cautions when using the COUNT function.

- The COUNT function performs additional processing other than the source code in the COBOL program, including collection of the COUNT information. Therefore, the size of the program increases, which may degrade the execution speed when using the COUNT function. Use the COUNT function only for debugging a program. After the debugging, specify the compiler option NOCOUNT to recompile the program.
- The output file at abnormal termination includes the statements that caused abnormal termination.
- When executing the CANCEL statement, the COUNT information of the program to be cancelled is output. If there is a program that is called from the program to be cancelled, the COUNT information of the program is generated by the calling program.
- You need to specify the environment variable SYSCOUNT to define the name of the output file.
- When an application called from a different language program terminates abnormally, the COUNT information might not be output.
- COUNT information is output to the output file specified for environment variable SYSCOUNT. It cannot be output from two or more processes to the same file at the same time - this will produce an output error at execution time. Change the output file name of each process when you execute two or more processes at the same time.

5.5 Using the Memory Check Function

The memory check function is used when area corruption occurs during COBOL application execution. This function checks the runtime system area at the start and end of a COBOL application procedure division. If one of the following execution time messages is output or an application error (segmentation violation) occurs, area corruption may have occurred, and an investigation with the memory check function must therefore be made into the cause of the area corruption.

- JMP0009I-U The library work area cannot be reserved (*1).
- JMP0010I-U The library work area is damaged.

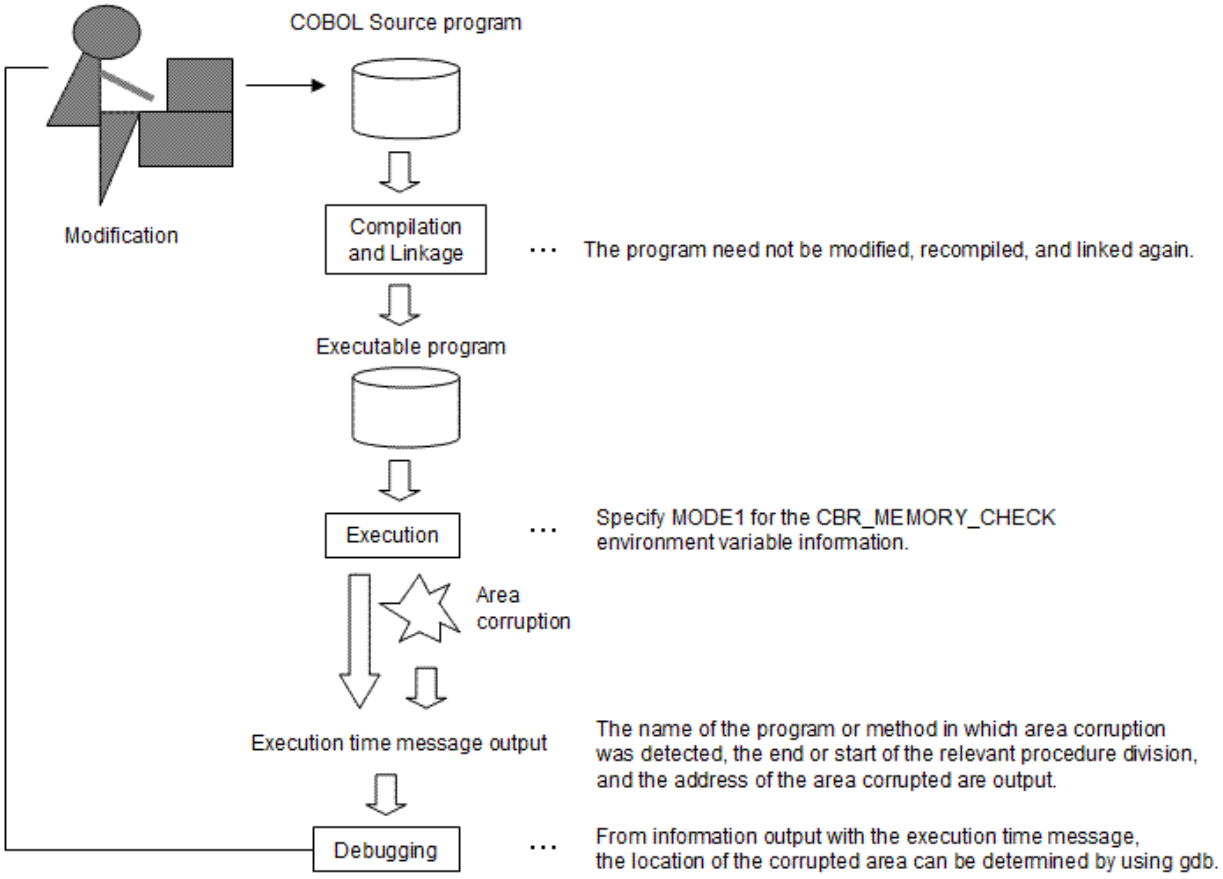
*1 : This message may be output even if there is insufficient virtual memory.

To use the memory check function, specify CBR_MEMORY_CHECK environment variable information.

```
$ CBR_MEMORY_CHECK=MODE1 ; export CBR_MEMORY_CHECK
```

5.5.1 Flow of debugging

The following figure shows the flow of debugging using the memory check function.



Identify programs in which areas have become corrupted using the information output in messages at the time of execution.

Use gdb to identify corrupted areas. Refer to "Chapter 18 Using the Debugger" for the method of using gdb.

5.5.2 Output messages

The memory check function outputs a message as follows when it detects memory corruption.

JMP0071I-U

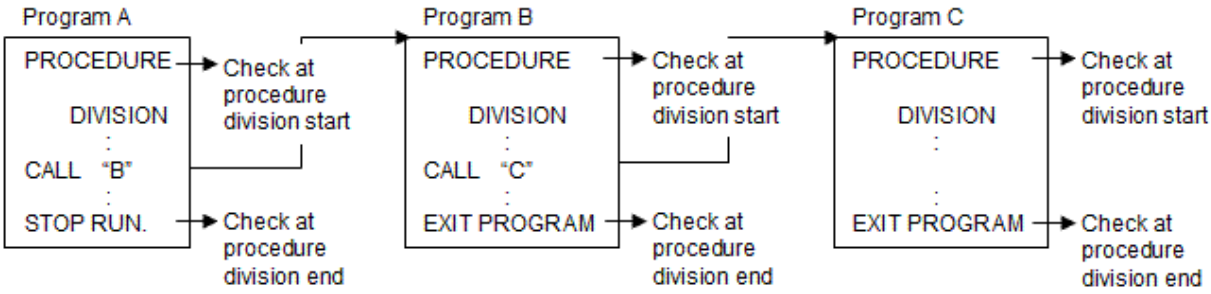
[PID:xxxxxxxx TID:xxxxxxxx] LIBRARY WORK AREA DESTRUCTION WAS DETECTED. $1 $2 BRKADR=0x$3.

Note that the location where the area destruction has been detected is set in $1, while the program or method name where the area destruction has been detected is set in $2. (The location is either START indicating the start of the program or that of the procedure of the method or END indicating the end.) The address of the area destroyed is set in $3.

5.5.3 Identifying a program

This section describes the method of identifying the program that caused area corruption.

The invocation relationship shown in the figure below is assumed for an output message indicating area damage.




```
JMP0071I-U [PID:00000010 TID:0000000E] LIBRARY WORK AREA DESTRUCTION WAS DETECTED.
                START PGM = C BRKADR = 0x00202000
```

The memory check function checks the runtime system area at the start and end of the procedure division of a program. In this example, area damage was not detected during the check at the procedure division start for COBOL program A or COBOL program B, and it was detected during the check at the procedure division start for COBOL program C. Therefore, the area damage is determined to have occurred after the check at the procedure division start for COBOL program B and before the check at the procedure division start for COBOL program C.

5.5.4 Note

The memory check function checks the runtime system area at the start and end of a program procedure division, thereby decreasing the execution speed. Disable the specification of the memory check function (CBR_MEMORY_CHECK environment variable information) after the end of debugging.

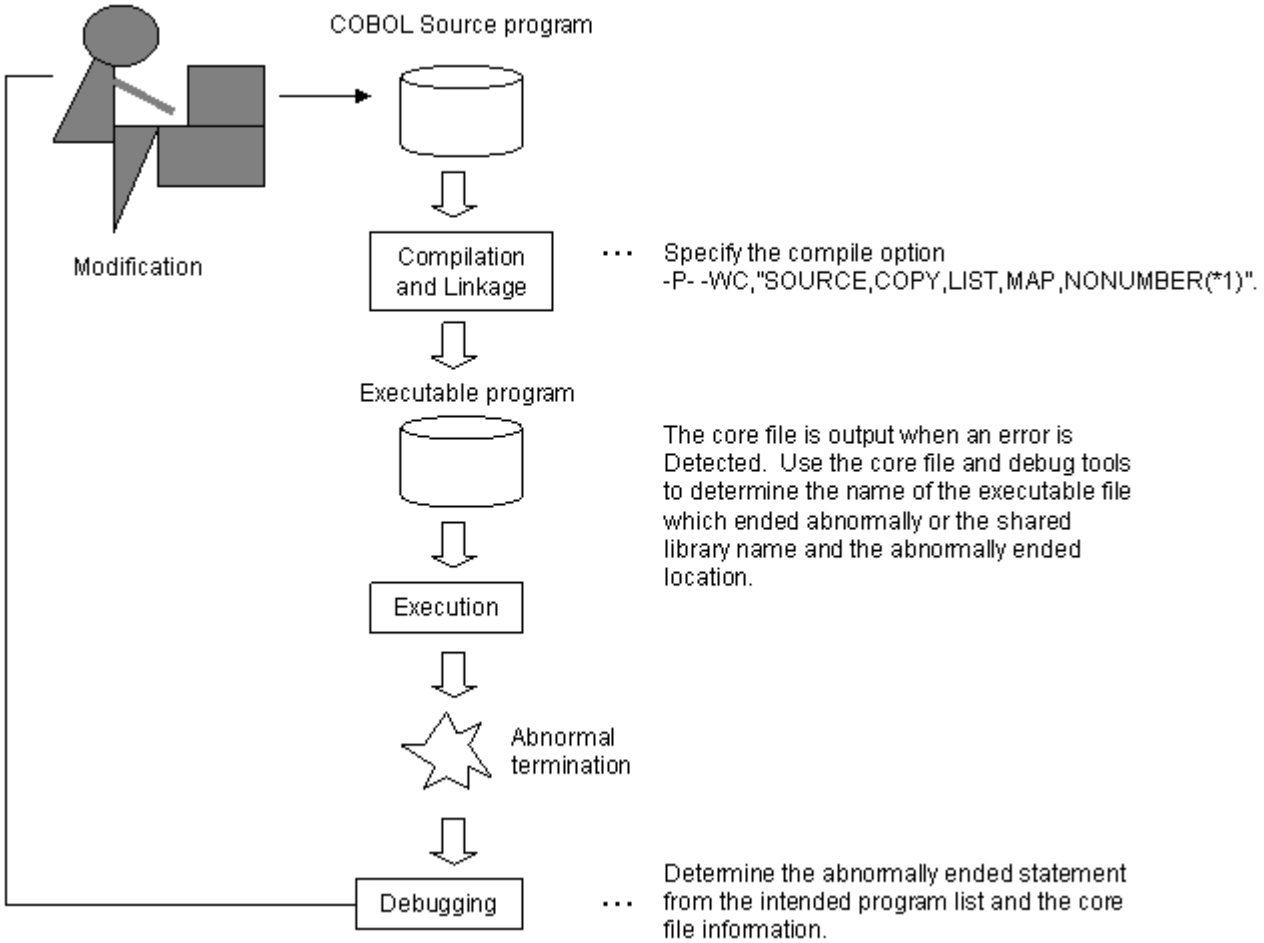
5.6 How to Identify the Portion Where an Error Occurred at Abnormal Termination

You can identify the portions in which errors occurred using an object program list and data map list other than the COBOL debugging function. This section explains about the object program list, including some examples and how to identify the portions where errors occurred.

5.6.1 The Flow of Debugging

The following is the flow of how to identify the portions where errors occurred using the object program list and data map list:

Figure 5.5 Debugging Flow



*1: If the compile option NUMBER is not explicitly specified, it is assumed that NONUMBER is specified. It is recommended that COBOL source be compiled with the compile option NONUMBER enabled. The explanations below assume that compilation is performed with NONUMBER enabled.

With the following compile options specified, acquire compile time information:

- SOURCE
- COPY (when the COPY library is used)
- MAP

If the OPTIMIZE option is enabled (default), also specify the following option to acquire the information:

- LIST

Example

```
cobol -c -WC,"NOOPTIMIZE,SOURCE,COPY,MAP" -P- DBGSUB.cob
cobol -M -WC,"NOOPTIMIZE,SOURCE,COPY,MAP" -P- DBGMAIN.cob DBGSUB.o
```

5.6.2 How to Identify the Portion Where an Error Occurred

Using a simple example, this section explains how to determine the location of a failure occurrence in cases where a program compiled with the "NOOPTIMIZE" option enabled terminates abnormally during execution. For details on how to determine the failure occurrence location of an abnormal termination when the OPTIMIZE option is enabled, see "18.2.6.2 Debugging using assembler."

1. Acquiring abnormal termination information

   If a program terminates abnormally during execution, for example, the following message is displayed:

```
$ ./a.out
Floating point exception (core dumped)
```

Note

If no core is generated, use shell commands such as the ulimit(bash)/limit(tcsh) command for referencing.

   Use a debugger to acquire information including the line where the abnormal termination occurred. The following is an example of the procedure for determining the line where the abnormal termination occurred.

   - Starting gdb

     The interrupted source location is output on the final line.

```
$ gdb a.out core.*
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Reading symbols from /opt/FJSVcbl64/lib/libFJBASE.so...done.
Loaded symbols for /opt/FJSVcbl64/lib/libFJBASE.so
Reading symbols from /opt/FJSVcbl64/lib/librcobol.so...done.
Loaded symbols for /opt/FJSVcbl64/lib/librcobol.so
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/libdl.so.2...done.
Loaded symbols for /lib64/libdl.so.2
Reading symbols from /opt/FJSVcbl64/lib/librcobflm64.so...done.
Loaded symbols for /opt/FJSVcbl64/lib/librcobflm64.so
Reading symbols from /lib64/libpthread.so.0...done.
Loaded symbols for /lib64/libpthread.so.0
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by `./a.out'.
Program terminated with signal 8, Arithmetic exception.
[New process 31709]
#0  0x00000000004017fa in DBGSUB () at DBGSUB.cob:19
19    000019     COMPUTE  AVE = TOTAL / DAYS
```

   - Displaying the source program

     If the source program is found in the environment being checked, the source program can be displayed.

```
(gdb) list
14      000014     PERFORM DAYS TIMES
15      000015         ADD  1  TO  IDX
16      000016         ADD  TMP-DAY(IDX)  TO  TOTAL
17      000017     END-PERFORM
18      000018*
```

```
19      000019     COMPUTE  AVE = TOTAL / DAYS
20      000020*
21      000021     EXIT PROGRAM
22      000022 END PROGRAM DBGSUB.
```

Note

- If the NUMBER compile option is enabled, the source program cannot be displayed correctly.
- For details on displaying the source program when the source program has not been found in the same path as that used at the compilation time, see the gdb document.

- Displaying data

  When data is displayed as is, each type of COBOL-specific data is displayed as a character string.

```
(gdb) print DAYS
$3 = "\000\000\000"
(gdb) print TOTAL
$1 = "00000000@"
(gdb) print AVE
$4 = "\000"
```

  To confirm the data excluding character strings, display data in hexadecimal notation.

```
(gdb) print/x DAYS
$2 = {0x0, 0x0, 0x0, 0x0}
(gdb) print/x TOTAL
$3 = {0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x40}
(gdb) print/x AVE
$1 = {0x0, 0x0}
```

  Use of a user definition command enables COBOL-specific data to be displayed.

```
(gdb) binary 4 DAYS
0 (0x00000000)
(gdb) zone 9 TOTAL
+0 (0x3030303030303040)
(gdb) binary 2 AVE
0 (0x0000)
```

  "DAYS" can be identified to be "0" in the divider data, so an abnormal termination occurred.

Note

For details on the user definition command, see "18.2.5.6 Displaying data (expression)."

- Checking the source program

  Check for the declaration location of "DAYS" in the source program list output during compilation.

```
      LINE   SEQNO  A   B

         1   000001 IDENTIFICATION DIVISION.
         2   000002 PROGRAM-ID. DBGSUB.
         3   000003 DATA DIVISION.
         4   000004 WORKING-STORAGE SECTION.
         5   000005 COPY EXTDATA.
       1-1 C 000001 01  ONE-WEEK USAGE DISPLAY IS EXTERNAL.
       1-2 C 000002  02  MON  PIC S9(3)V9.
       1-3 C 000003  02  TUE  PIC S9(3)V9.
       1-4 C 000004  02  WED  PIC S9(3)V9.
```

```
        1-5 C 000005  02  THU  PIC S9(3)V9.
        1-6 C 000006  02  FRI  PIC S9(3)V9.
        1-7 C 000007  02  SAT  PIC S9(3)V9.
        1-8 C 000008  02  SUN  PIC S9(3)V9.
        1-9 C 000009 01 REDEFINES ONE-WEEK.
       1-10 C 000010  02  TMP-DAY  PIC S9(3)V9 USAGE DISPLAY OCCURS 7 TIMES.
          6   000006 01  TOTAL PIC S9(8)V9 USAGE DISPLAY.
          7   000007 77  IDX   PIC S9(9)   USAGE COMP-5.
          8   000008 LINKAGE SECTION.
          9   000009 01  DAYS  PIC 9(9)    USAGE BINARY.
         10   000010 01  AVE   PIC S9(3)V9 USAGE BINARY.
         11   000011 PROCEDURE DIVISION USING DAYS RETURNING AVE.
         12   000012     MOVE  0  TO  TOTAL
         13   000013     MOVE  0  TO  IDX
         14   000014     PERFORM DAYS TIMES
         15   000015         ADD  1  TO  IDX
         16   000016         ADD  TMP-DAY(IDX)  TO  TOTAL
         17   000017     END-PERFORM
         18   000018*
         19   000019     COMPUTE  AVE = TOTAL / DAYS
         20   000020*
         21   000021     EXIT PROGRAM
         22   000022 END PROGRAM DBGSUB.
```

"DAYS" is an argument transferred from the calling program.

- Displaying the calling path (back trace)

  Display the calling path to acquire the location of the calling program.

```
(gdb) backtrace
#0  0x4000000000003d10 in DBGSUB () at DBGSUB.cob:19
#1  0x4000000000001e80 in DBGMAIN () at DBGMAIN.cob:18
#2  0x4000000000000c70 in DBGMAIN () at DBGMAIN.cob:1
```

Note

If an internal subprogram is found in the calling path, the correct calling path cannot be displayed. In such cases, change the internal subprogram to an external program temporarily and perform debugging.

- Checking the calling source program

  Check for the flow of the "DAYS" data in the source program list of the calling program.

```
      LINE   SEQNO  A   B

         1   000001 IDENTIFICATION DIVISION.
         2   000002 PROGRAM-ID. DBGMAIN.
         3   000003 DATA DIVISION.
         4   000004 WORKING-STORAGE SECTION.
         5   000005 01  DAYS     PIC 9(9)    USAGE BINARY.
         6   000006 COPY EXTDATA.
       1-1 C 000001 01  ONE-WEEK USAGE DISPLAY IS EXTERNAL.
       1-2 C 000002  02  MON  PIC S9(3)V9.
       1-3 C 000003  02  TUE  PIC S9(3)V9.
       1-4 C 000004  02  WED  PIC S9(3)V9.
       1-5 C 000005  02  THU  PIC S9(3)V9.
       1-6 C 000006  02  FRI  PIC S9(3)V9.
       1-7 C 000007  02  SAT  PIC S9(3)V9.
       1-8 C 000008  02  SUN  PIC S9(3)V9.
       1-9 C 000009 01 REDEFINES ONE-WEEK.
      1-10 C 000010  02  TMP-DAY  PIC S9(3)V9 USAGE DISPLAY OCCURS 7 TIMES.
         7   000007 01  AVE      PIC S9(3)V9 USAGE BINARY.
```

```
  8   000008 01  AVE-TEMP PIC ++9.9   USAGE DISPLAY.
  9   000009 PROCEDURE DIVISION.
 10   000010     MOVE  4.0  TO  MON
 11   000011     MOVE  0.0  TO  TUE
 12   000012     MOVE -0.2  TO  WED
 13   000013     MOVE  2.5  TO  THU
 14   000014     MOVE  5.0  TO  FRI
 15   000015     MOVE  6.5  TO  SAT
 16   000016     MOVE  5.0  TO  SUN
 17   000017*
 18   000018     CALL "DBGSUB" USING DAYS RETURNING AVE
 19   000019*
 20   000020     MOVE AVE TO AVE-TEMP
 21   000021     DISPLAY "AVERAGE TEMPERATURE OF THIS WEEK IS" AVE-TEMP "DEGREES."
 22   000022     EXIT PROGRAM
 23   000023 END PROGRAM DBGMAIN.
```

"0" is displayed because, as can be identified, no value is set for "DAYS." The program runs correctly when a correct value is set.

2. Displaying data by using the data map list

   Data can be referenced by using the "Data map list" of the "Data area-related list" and "Program control information list".

Note

- Only data used by the abnormally terminating program or method can be referenced. Data for another program or another method cannot be referenced.
- If the OPTIMIZE compile option is enabled (default), execution results may not be saved in the area because of optimization processing. Consequently, the data may not be able to be referenced correctly with the above method.

- Explanation of data areas

  The four basic data areas are as follows:

  - stack data
  - heap data
  - .data
  - .rodata

  The following additional data areas are also used for method definitions:

  - .rodata used by object data or factory data
  - Heap data used by object data or factory data

Data area	Notation		Explanation
	*1	*2	
stack data	stac	stack	The stack data area stores data entered in the working-storage section of method definitions. It also stores work data and management data for compiler generation. These areas extend from high (large) addresses to low (small) addresses. The "rsp" register manages the boundaries of used areas and unused areas. The "rsp" register is referenced when COBOL procedures are executed.

Data area	Notation		Explanation
	*1	*2	
heap data	heap	heap	The heap data area stores data entered in the working-storage section of program definitions. It also stores work data and management data for compiler generation. The "rbx" register is referenced when COBOL procedures are executed.
.data	data	.data	The .data area is the read-enabled and write-enabled data area. This area stores work data and management data for compiler generation.
.rodata	rdat	.rodata	The .rodata area is the read-specific data area. This area stores data entered in the constants section, and read-specific data for compiler generation and other data that does not change during procedure execution.
.rodata used by object data or factory data	-	O/F .rodata	This area stores read-specific shared data in object definitions or factory definitions.
Heap data used by object data or factory data	hea2	O/F heap	This area stores shared data in object definitions or factory definitions.

*1 : This notation is used in the "address" part explained in the data map list output format described in "3.1.7.6 Data Area Listings".

*2 : This notation is used in the "Figure 5.6 The CONTROL ADDRESS TABLE (control information table) structure" explanations.

Use the following method to obtain the start of each of the data areas:

[Program control information listing (DBGSUB.lst)]

```
     ADDR              FIELD-NAME                      LENGTH
:
    ** HGWA **
      * HGCB *
heap+00000000          HGCB FIXED AREA                    72
      * HGMB *
heap+00000048          LIA FIXED AREA                    256
     ........          IWA1 AREA                           0
     ........            LINAGE COUNTER                    0
     ........            OTHER AREA                        0
     ........            ALTINX                            0
     ........            PCT                               0
heap+00000148          LCB AREA                           80
heap+00000198          HGMB POINTERS AREA                 80
     ........            VPA                               0
     ........            PSA                               0
heap+00000198            BVA                               8
heap+000001A0            BEA                               8
     ........            FMBE                              0
heap+000001A8            CONTROL ADDRESS TABLE            64      ... [1]
     ........            MUTEX HANDLE AREA                 0
:
    ** STK **
      * SCB *
stac+00000000          ABIA                               24
stac+00000020          SCB FIXED AREA                    112
stac+00000090          TL 1ST AREA                        48
```

```
stac+000000C0          LCB AREA                          80
stac+00000110          SGM POINTERS AREA                  8
     ........            VPA                              0
     ........            PSA                              0
     ........            BVA                              0
stac+00000110            BHG                              8      ... [2]
     ........            BOD                              0
```

[Obtaining the start address of the heap area]

The start address of the heap area is set in the "rbx" register when a procedure is executed. At times other than procedure execution, the start address of the heap area is not set in the "rbx" register

The method below can be used to check whether or not "the "rbx" register value = heap area start address".

- Check whether the value is the same as the value set at [2], BHG(stac+00000110).

Since "stac" is set in "rsp", use gdb to display the "rsp+0x110" memory contents and the "rbx" register contents and compare them.

[rbx register contents]

```
(gdb) info register rbx
rbx            0x8f6f520        150402336
```

[rsp+0x110 memory contents]

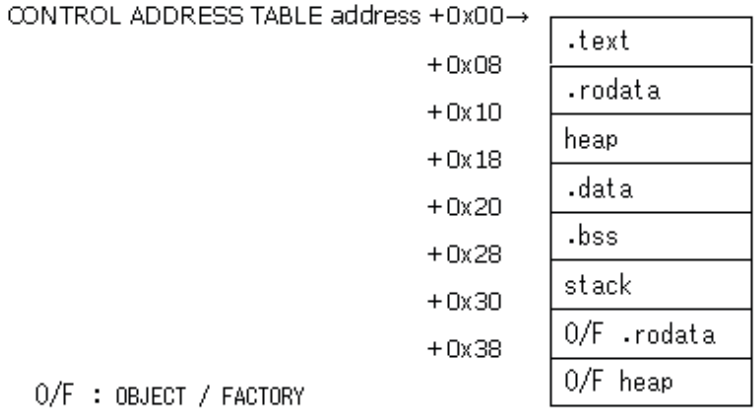
```
(gdb) x/gx $rsp+0x110
0x7fffb66efaa0: 0x0000000008f6f520
```

If they match as shown above, the "rbx" register value as is can be used as the heap area start address.

[Obtaining the start address of each of the data areas]

Once the heap area start address is obtained, the contents of the CONTROL ADDRESS TABLE (control information table) ([1])are displayed to obtain the start address of each of the data areas.

Figure 5.6 The CONTROL ADDRESS TABLE (control information table) structure



In this example, [1], the CONTROL ADDRESS TABLE, is at "heap+000001A8". Therefore, check the contents of this location in memory. Here, it is assumed that "the "rbx" register value = heap area start address", and "rbx" can be substituted for "heap" in the calculation.

NetCOBOL provides the user definition command "CAT" to output the contents of the CONTROL ADDRESS TABLE.

[CONTROL ADDRESS TABLE memory contents]

```
(gdb) CAT $rbx+0x1a8
                .text
0x8f6f6c8:      0x00000000004010c0
```

```
                .rodata
0x8f6f6d0:      0x0000000000401e60
                heap
0x8f6f6d8:      0x0000000008f6f520
                .data
0x8f6f6e0:      0x0000000000602610
                .bss
0x8f6f6e8:      0x0000000000000000
                stack
0x8f6f6f0:      0x00007fffb66ef990
                O/F .rodata
0x8f6f6f8:      0x0000000000000000
                O/F heap
0x8f6f700:      0x0000000000000000
```

[Data map listing (DBGSUB.lst)]

```
LINE   ADDR    OFFSET REC-OFFSET LVL NAME  LENGTH ATTRIBUTE  BASE      DIM
 1-1 [heap+000001A0]+00000000    0 01 ONE-WEEK  28 GROUP-F    BEA.000001
 1-2 [heap+000001A0]+00000000    0 02 MON        4 EXT-DEC    BEA.000001
 1-3 [heap+000001A0]+00000004    4 02 TUE        4 EXT-DEC    BEA.000001
 1-4 [heap+000001A0]+00000008    8 02 WED        4 EXT-DEC    BEA.000001
 1-5 [heap+000001A0]+0000000C   12 02 THU        4 EXT-DEC    BEA.000001
 1-6 [heap+000001A0]+00000010   16 02 FRI        4 EXT-DEC    BEA.000001
 1-7 [heap+000001A0]+00000014   20 02 SAT        4 EXT-DEC    BEA.000001
 1-8 [heap+000001A0]+00000018   24 02 SUN        4 EXT-DEC    BEA.000001
 1-9 [heap+000001A0]+00000000    0 01 FILLER    28 GROUP-F    BEA.000001
1-10 [heap+000001A0]+00000000    0 02 TMP-DAY    4 EXT-DEC    BEA.000001 1
   6  heap+000001F0              0 01 TOTAL      9 EXT-DEC    BHG.000000
   7  heap+000001FC              0 77 IDX        4 COMP-5     BHG.000000
   9 [heap+00000198]+00000000    0 01 DAYS       4 BINARY     BVA.000001
  10  stac+000001D0              0 01 AVE        2 BINARY     BST.000000
```

Proceed as shown below to reference the "Wednesday" data.

```
(gdb) zone 4 'WED'
-2 (0x30303052)
```

Below, the "address" and "offset" information displayed in the data map list is used in the memory display (x) subcommand to display "Wednesday".

Since the start address for "heap" is stored in the "rbx" register, "rbx" can be substituted for "heap" in the calculation.

```
(gdb) x/4bx {long}($rbx+0x1a0)+8
0x8f6dda8:      0x30    0x30    0x30    0x52
```

Proceed as shown below to reference the "Wednesday" data redefined by the REDEFINES clause.

Add the offset "8", from the data start to the third element ("Wednesday" data), to the data "Day" in the OCCURS clause specification to display the "Wednesday" data.

```
(gdb) zone 4 'TMP-DAY'+8
-2 (0x30303052)
```

Information

The "cobcorechk" command is a user definition command provided for output of abnormal statuses together.

The "cobcorechkout" command used to output results to a file is also provided.

The results are overwritten in the "cobcorechk.txt" file in the directory in which gdb was started.

Additionally, the following commands are provided for acquiring more detailed information:

- "cobcorechkex"

  Outputs detailed COBOL runtime environment information.

- "cobcorechkexlist"

  Outputs COBOL runtime environment information in the form of a simple list.

- "cobcorechkexout"

  Overwrites the "cobcorechkex" results in the "cobcorechkex.txt" file during saving.

- "cobcorechkexlistout"

  Overwrites the "cobcorechkexlist" results in the "cobcorechkexlist.txt" file during saving.

Use the cobcorechkout command to output to file the status at the time of the error.

```
(gdb) cobcorechkout

*****************************************************************************
* Start of cobcorechk
*****************************************************************************
* Program information
*****************************************************************************
Using the running image of child Thread 0x2ada4a85f0c0 (LWP 12895).
Program stopped at 0x4017be.
It stopped with signal SIGFPE, Arithmetic exception.
Type "info stack" or "info registers" for more information.

*****************************************************************************
* Environment variable
*****************************************************************************
MANPATH=/opt/FJSVcbl64/man/%L:/opt/FJSVcbl64/man:/opt/FJSVXbsrt/man/%L:/opt/FJSVXbsrt/
man:/usr/share/man/%L:/usr/share/man:/usr/X11R6/man:/usr/local/man:/usr/man/%L:/usr/
man:/usr/kerberos/man
HOSTNAME=x64lintest
TERM=vt100
SHELL=/bin/bash
HISTSIZE=1000
KDE_NO_IPV6=1
SSH_CLIENT=10.124.40.232 2191 22
QTDIR=/usr/lib64/qt-3.3
OLDPWD=/home/user
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/1
COBCOPY=/opt/FJSVcbl64/copy:.
USER=comp
LD_LIBRARY_PATH=/opt/FSUNiconv/lib:.:/opt/FJSVcbl64/lib:/opt/FJSVXbsrt/lib:/opt/FJSVXmeft/
lib:/opt/FJSVcbl64/lib
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01;05;3
7;41:mi=01;05;37;41:ex=01;32:*.cmd=01;32:*.exe=01;32:*.com=01;32:*.btm=01;32:*.bat=01;32:*.s
h=01;32:*.csh=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.zip=01;31:
*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.bz=01;31:*.tz=01;31:*.rpm=01;31:*.cpio=01;31:*.
jpg=01;35:*.gif=01;35:*.bmp=01;35:*.xbm=01;35:*.xpm=01;35:*.png=01;35:*.tif=01;35:
KDEDIR=/usr
NLSPATH=/opt/FJSVcbl64/lib/nls/%L/%N.cat:/opt/FJSVcbl64/lib/nls/C/%N.cat:/opt/
FJSVXbsrt/lib/nls/%L/%N.cat:/opt/FJSVXbsrt/lib/nls/C/%N.cat:/opt/FJSVcbl64/lib/nls/%L/%N.cat
MAIL=/var/spool/mail/comp
PATH=/opt/FJSVcbl64/bin:/opt/FJSVXbsrt/bin:/usr/bin:/bin:/opt/FJSVcbl64/bin:/usr/lib64/
qt-3.3/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/sbin:/sbin:/home/comp/bin
INPUTRC=/etc/inputrc
PWD=/home/user
LANG=C
KDE_IS_PRELINKED=1
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
```

```
SHLVL=1
HOME=/home/comp
LOGNAME=comp
QTLIB=/usr/lib64/qt-3.3/lib
CVS_RSH=ssh
SSH_CONNECTION=10.124.40.232 2191 10.124.40.209 22
LESSOPEN=|/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
_=/usr/bin/gdb
LINES=35
COLUMNS=146

*****************************************************************************
* Shared library
*****************************************************************************
From                To                  Syms Read   Shared Object Library
0x0000003620a00a70  0x0000003620a166fe  Yes         /lib64/ld-linux-x86-64.so.2
0x00002ada4a0c7960  0x00002ada4a0ca0d0  Yes
/opt/FJSVcbl64/lib/libFJBASE.so
0x00002ada4a2d47c0  0x00002ada4a39b35d  Yes
/opt/FJSVcbl64/lib/librcobol.so
0x0000003620e1d760  0x0000003620f08ed8  Yes         /lib64/libc.so.6
0x0000003621600e10  0x0000003621601a08  Yes         /lib64/libdl.so.2
0x00002ada4a62e600  0x00002ada4a6540d8  Yes
/opt/FJSVcbl64/lib/librcobflm64.so
0x0000003621a051f0  0x0000003621a0fb68  Yes         /lib64/libpthread.so.0

000000001dd209f0
000000001dd209f0 /opt/FJSVcbl64/lib/libFJBASE.so
000000001dd209f0 /opt/FJSVcbl64/lib/librcobol.so
000000001dd209f0 /lib64/libc.so.6
000000001dd209f0 /lib64/libdl.so.2
000000001dd209f0 /opt/FJSVcbl64/lib/librcobflm64.so
000000001dd209f0 /lib64/libpthread.so.0
000000001dd209f0 /lib64/ld-linux-x86-64.so.2

*****************************************************************************
* Thread
*****************************************************************************
* 1 Thread 0x2ada4a85f0c0 (LWP 12895)  0x00000000004017be in DBGSUB () at DBGSUB.cob:19

*****************************************************************************
* Register
*****************************************************************************
rax            0x0      0
rbx            0x159c24c0       362554560
rcx            0x6025c0 6301120
rdx            0x0      0
rsi            0x159c26b0       362555056
rdi            0x7fff609e1ac4   140734814362308
rbp            0x7fff609e1ce0   0x7fff609e1ce0
rsp            0x7fff609e1a20   0x7fff609e1a20
r8             0x8      8
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x401e20 4202016
r13            0x159b5280       362500736
r14            0x159c0c50       362548304
r15            0x0      0
rip            0x4017be 0x4017be <DBGSUB+1798>
eflags         0x10287  [ CF PF SF IF RF ]
cs             0x33     51
```

```
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
mxcsr          0x1f80   [ IM DM ZM OM UM PM ]

*************************************************************************
* Instruction
*************************************************************************
IP=0x00000000004017be

0x4017be <DBGSUB+1798>: idiv   %r9
0x4017c1 <DBGSUB+1801>: mov    %rax,%r11
0x4017c4 <DBGSUB+1804>: mov    %r11,%rax
0x4017c7 <DBGSUB+1807>: rol    $0x8,%ax
0x4017cb <DBGSUB+1811>: mov    %ax,0x1d0(%rsp)
0x4017d3 <DBGSUB+1819>: mov    $0x0,%r11
0x4017da <DBGSUB+1826>: mov    %r11d,0x108(%rsp)
0x4017e2 <DBGSUB+1834>: mov    $0x15,%r11
0x4017e9 <DBGSUB+1841>: mov    %r11d,0x10c(%rsp)
0x4017f1 <DBGSUB+1849>: jmpq   0x4013e4 <DBGSUB+812>

*************************************************************************
* Back trace
*************************************************************************
#0  0x00000000004017be in DBGSUB () at DBGSUB.cob:19
#1  0x0000000000400d79 in DBGMAIN () at DBGMAIN.cob:18
#2  0x0000000000400910 in main () at DBGMAIN.cob:1

*************************************************************************
* Local data
*************************************************************************
PROGRAM-STATUS = 0
RETURN-CODE = 0
TALLY = 0
ONE-WEEK = "004@000@000R002E005@006E005@"
MON = "004@"
TUE = "000@"
WED = "000R"
THU = "002E"
FRI = "005@"
SAT = "006E"
SUN = "005@"
TMP-DAY = "004@"
TOTAL = "00000000@"
IDX = 0
DAYS = "\000\000\000"
AVE = "\000"

*************************************************************************
* Source list
*************************************************************************
0x4017be is in DBGSUB (DBGSUB.cob:19).
4       000004 WORKING-STORAGE SECTION.
```

```
5       000005 COPY EXTDATA.
6       000006 01  TOTAL PIC S9(8)V9 USAGE DISPLAY.
7       000007 77  IDX   PIC S9(9)   USAGE COMP-5.
8       000008 LINKAGE SECTION.
9       000009 01  DAYS  PIC 9(9)    USAGE BINARY.
10      000010 01  AVE   PIC S9(3)V9 USAGE BINARY.
11      000011 PROCEDURE DIVISION USING DAYS RETURNING AVE.
12      000012     MOVE  0  TO  TOTAL
13      000013     MOVE  0  TO  IDX
14      000014     PERFORM DAYS TIMES
15      000015         ADD  1  TO  IDX
16      000016         ADD  TMP-DAY(IDX)  TO  TOTAL
17      000017     END-PERFORM
18      000018*
19      000019     COMPUTE  AVE = TOTAL / DAYS
20      000020*
21      000021     EXIT PROGRAM
22      000022 END PROGRAM DBGSUB.

*****************************************************************************
* End of cobcorechk
*****************************************************************************
```

Chapter 6 File Processing

This chapter offers you information on how NetCOBOL processes files, including file types and characteristics, record design, and file processing methods. This chapter also details the differences between record sequential, line sequential, relative and indexed files, including usage and processing information.

Additionally, information describing input-output error processing, file allocation and control improving file processing performance and using high-speed processing is also provided.

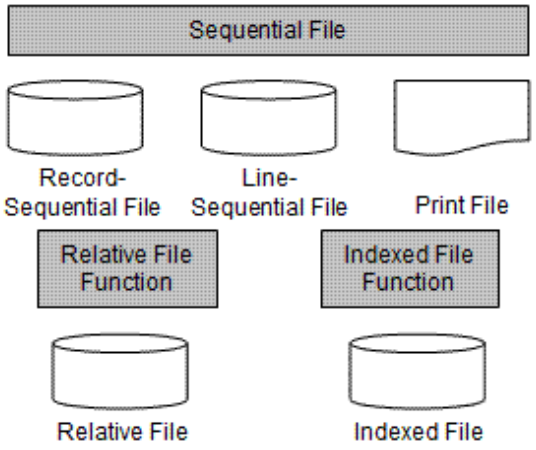
6.1 File Organization Types

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

The files in the following diagram can be processed using the sequential, relative, and indexed file functions:

Figure 6.1 File Organizations



The following table shows characteristics of each file type:

Table 6.1 File organization types and characteristics

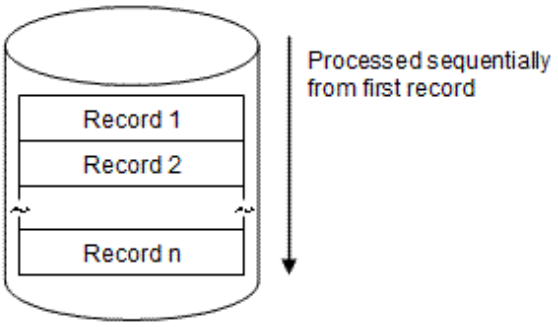
File Types	Record-Sequential File	Line-Sequential File	Print File	Relative File	Indexed File
Record processing	Record storage sequence			Relative record number	Record key value
Usable data medium	Hard disk (*1)	Hard disk (*1)	Printer Hard disk (*1)	Hard disk (*1)	Hard disk (*1)
Usage example	Saving data Work file	Text file	Printing data	Work file	Master file

*1 : Virtual devices (/dev/null and similar) cannot be used.

File organization type 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 type appropriate for its use. Each file organization is explained below.

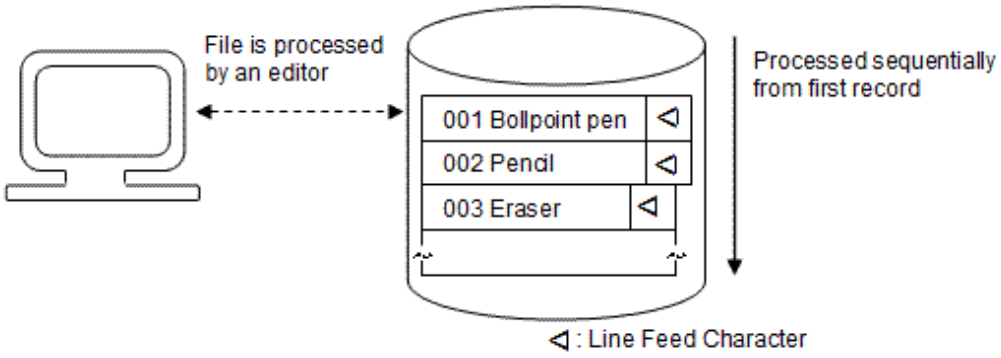
Record Sequential Files

In a record sequential file, records are read and updated in the order they are written, 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

In a line sequential file records are read in the order they are written, from the beginning of the file. Line feed characters are used as record delimiters in a line sequential file. For example, line sequential files can be used to handle files created by text editors.



The line feed character is 1 byte in size. The content of the line feed character is shown by the hexadecimal number mark.

```
0x0A
```

Note

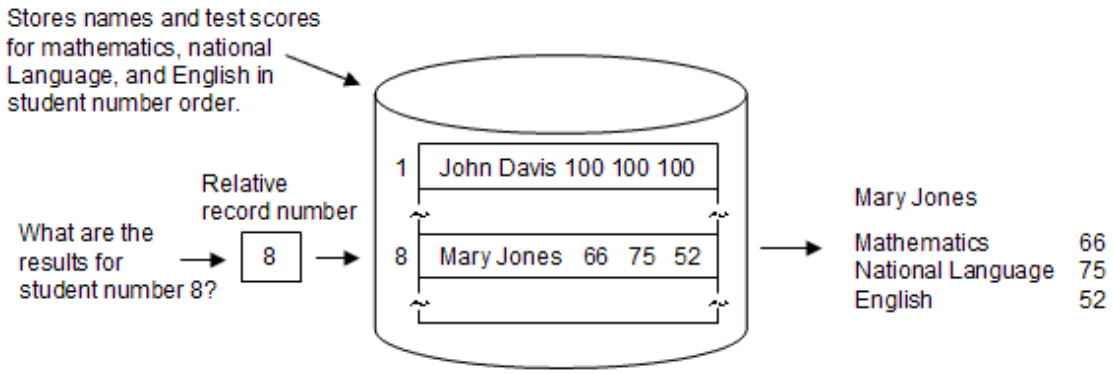
- For more information regarding the line feed character, refer to "Handling a control character in a record".
- When a WRITE statement with an ADVANCING clause is executed, non-line feed control characters are written. For more details, refer to "Operation with ADVANCING clause".

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 more details about a print file, refer to "Chapter 7 Printing".

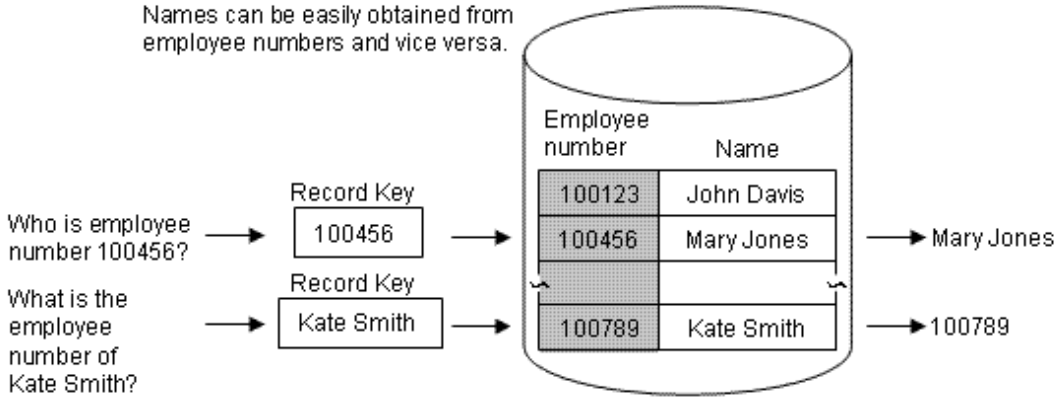
Relative Files

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

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 when using 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, one record is delimited by a constant record length, and the size of the record under the file is all the same.

Variable length record

In a variable length record, record-size is different in each record. The record size is determined when a record is written to the 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 and alternate record 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

When determining record keys, the following points must be considered:

- 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 6.2 File organization types and processing

File Types	Access Mode	Processing					
		Creation	Extension	Insertion	Reference	Updating	Deletion
Record sequential file	Sequential	A	A	B	A	A	B
Line sequential file	Sequential	A	A	B	A	B	B
Print file	Sequential	A	A	B	B	B	B
Relative file Indexed file	Sequential	A	A	B	A	A	A
	Random	A	B	A	A	A	A
	Dynamic	A	B	A	A	A	A

A: Can be processed

B: Cannot be processed.

When processing files, you can restrict access to the files in either of the following two ways: putting the files in exclusive mode or locking the records used by the file. This is called locking control of files. For details of locking control of files, see "6.7.3 Locking Control of Files".

6.2 Using Record Sequential Files

This section explains the following record sequential file processes:

- Defining record sequential files.
- Defining the records.
- Processing record sequential files.

```
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 entries
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, and then write it in the SELECT clause. Determine a file-reference-identifier, and declare 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 methods to determine what to specify for the file-reference-identifier are as follows:

- If 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.
- If the file name of the input-output medium is undetermined at COBOL program generation, or to determine the file name at program execution, specify a file identifier.
- To determine the file name in the program, specify a data name.
- Specify the DISK character string for temporary files that are not required after the program ends.

Note

Even if the DISK character string is specified for it, a file being used is not deleted when the program ends.

The file name of an actual input-output medium can include the following characters:

```
Blank spaces, the plus sign (+), commas (,), semi-colons (;), the equals sign (=), square brackets
([ ]), parentheses ( ), and single-quotation marks (').
```

Any file name that includes a comma (,) must be enclosed in double quotation marks (" ").

Information

Files can be processed at high speed in record sequential or line sequential files. For the specification method, refer to "High-speed File Processing" in this chapter.

The following table list examples of SELECT and ASSIGN clauses:

Table 6.3 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 "/home/data"`	-
DISK	`SELECT data1` `ASSIGN TO DISK`	- If describing file names in lower-case alphabetical characters, specify the compiler option NOALPHAL at compilation. - If the compiler option ALPHAL is available, process DATA1 in the current directory. - 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

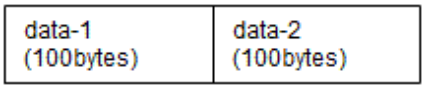
Record sequential files are either fixed or variable length. The record length in fixed length record format is the value specified in the RECORD clause or, if the RECORD clause is omitted, it is the length of the longest record defined in the record description entries.

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

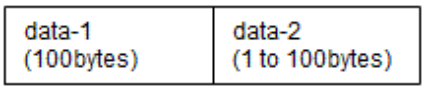
Use record description entries to define the attributes, positions, and sizes of the data in a record. The following are examples of record definitions:

- Fixed length record



```
FD   data-storage-file.
01   data-record.
      02  data-1  PIC X(100).
      02  data-2  PIC X(100).
```

- Variable length record




```
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.
*>       :
 WORKING-STORAGE SECTION.
 01   record-length      PIC 9(3) BINARY.
 01   length-1           PIC 9(3) BINARY.
*>       :
```

Note

Note the following points about the variable length record format that is configured from several data items contained in variable length data items containing the OCCURS clause.

Writing records

To move data to a record item, you must move data and the data length from the first variable length data item in order. There might be an unintentional result, depending on the order in which the data is moved.

Reading records

When you read a record, although length of the record you have read is returned, the length of the variable length data items contained in the record is not returned. For this reason, subtract the fixed part of the length from the length of the whole record, to find the length of the variable length data item. If several variable length data items have been defined, you cannot find out the length using this calculation. Instead, mount the information that shows the end of the variable part in the record, or define the fields of length of each variable length data item.

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 in record 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, execute the OPEN statement in the OUTPUT mode (OUTPUT phrase).

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.

Creation

To generate a record sequential file, open a file in OUTPUT mode and write records to the file. If an attempt is made to generate a file that already exists, the file is regenerated and the original contents are lost.

Perform the following procedure to create a file:

```
OPEN OUTPUT  file-name.
 Editing-records
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the OUTPUT phrase in an OPEN statement.
2. Set the record contents to be written.
3. Write the record with a WRITE statement.
4. To write all the records, repeat steps 2 and 3, and then close the file with a CLOSE statement.

Extension

To extend a sequential file, open the file in EXTEND mode, then write records to the file. In this case, records are added after the last record in the file.

Perform the following procedure to extend a file:

```
OPEN EXTEND  file-name.
 Editing-records
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the EXTEND phrase in an OPEN statement.
2. Set the record contents to be written.
3. Write the record with a WRITE statement.
4. To write all records, repeat steps 2 and 3, and then close the file with a CLOSE statement.

Reference

To refer to records, open the file in INPUT mode, then read records in the file, starting with the first record.

Perform the following procedure to refer a file:

```
OPEN INPUT  file-name.
READ   file-name  ... .
CLOSE  file-name.
```

1. Open a file with the INPUT phrase in an OPEN statement.
2. Read the records starting with the first record with a READ statement.
3. To read all the necessary records, repeat step 2, then close the file with a CLOSE statement.

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 READ statement. For more information on the at end condition, refer to "6.6 Input-Output Error Processing".

Update

To update records, open the file in I-O mode, then rewrite file records.

Perform the following procedure to update files:

```
OPEN I-O  file-name.
READ  file-name  ... .
 Editing-records
REWRITE  record-name  ... .
CLOSE    file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Read the records starting with the first record with a READ statement.
3. Update the read record contents.
4. Write updated records with a REWRITE statement.
5. To update all the necessary records, repeat steps 2 to 4, then close the file with a CLOSE statement.

- If 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 the following points about line sequential files:

- Defining line sequential files.
- Defining the records.
- Processing line sequential files.

```
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 entries
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, refer to "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.

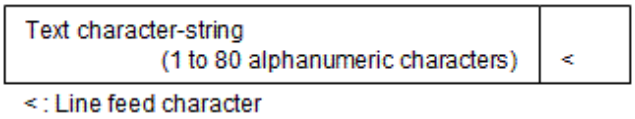
Record Formats and Lengths

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" in this chapter 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.

An example of record definitions in the variable length records format




```
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 for input and output 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 phrase).

Other Statements

To read records in a file, use a READ statement.

To write records to a file, use a WRITE statement.

Creation

To create a line sequential file, open a file in OUTPUT mode, then write records to the file. If an attempt is made to create a file that already exists, the file is recreated and the original contents are lost.

Perform the following procedure to create files:

```
OPEN OUTPUT  file-name.
 Editing-records
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the OUTPUT phrase in an OPEN statement.
2. Set the record contents to be written.
3. Write the record with a WRITE statement.
4. To write all the records, repeat step 2 and 3, then close the file with a CLOSE statement.

- 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 more details, refer to "6.8.2.2 Specification of Trailing Blanks in the Line Sequential File".

Extension

To extend a sequential file, open the file in EXTEND mode, and then write records to the file. In this case, records are added after the last record in the file.

Perform the following procedure to extend files:

```
OPEN EXTEND  file-name.
 Editing-records
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the EXTEND phrase in an OPEN statement.
2. Set the record contents to be written.
3. Write the record with a WRITE statement.
4. To write all the records, repeat steps 2 and 3, and then close the file with a CLOSE statement.

To output the record and remove trailing blanks in the record, specify character string "REMOVE" in environment variable CBR_TRAILING_BLANK_RECORD. For more details, refer to "6.8.2.2 Specification of Trailing Blanks in the Line Sequential File".

Reference

To refer to records, open the file in INPUT mode, then read records in the file, starting with the first record.

Perform the following procedure to refer files:

```
OPEN INPUT  file-name.
READ   file-name  ... .
CLOSE  file-name.
```

1. Open a file with the INPUT phrase in an OPEN statement.
2. Read the records, starting with the first record, with a READ statement.
3. To read all the necessary records, repeat step 2, then close the file with a CLOSE statement.

Note

- If 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 READ statement.

  For more information on the at end condition, refer to "6.6 Input-Output Error Processing".

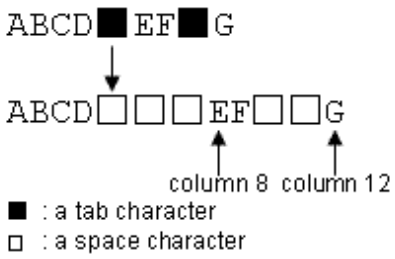
- If the size of a read record is greater than the record length, data with the same length as the record length is set in the record area when a READ statement is executed. The continuation of the data of the same record is then set in the record area when the next READ statement is executed. If the size of data to be set is smaller than the record length, spaces are written to the remaining portion of the record area.

Treatment of TAB

If there are any tab characters in an input record, spaces are inserted to align as follows:

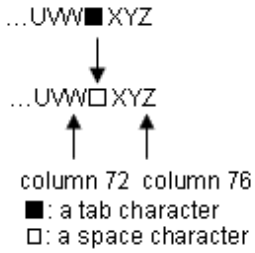
If tab characters are present up to character position 72

The first character position is assumed as 1, in order to align the characters after tab characters spaces are inserted at the following character positions 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, and 72.



If tab characters are present after character position 72

Tab characters are read as single spaces.



Handling a control character in a record

A control character contained in a record to be read is treated as follows:

- 0x0C (page feed): Treated as a record delimiter
- 0x0D (return): Treated as a record delimiter
- 0x1A (data end symbol): Treated as a file terminator

The meaning of a control character is enclosed between parentheses.

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 } 0x0A
BEFORE	n LINES (*1)	{ record-data } 0x0A ··· 0x0A (n)
	PAGE	{ record-data } 0x0C
AFTER	n LINES (*1)	0x0A ··· 0x0A { record-data } 0x0D (n)
	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.

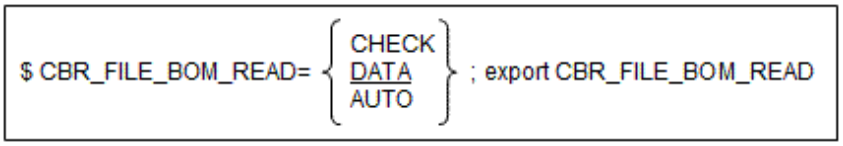
  0x0D 0x0A
  0x0D 0x0C
  0x0A 0x0D
  0x0C 0x0D

Refer to "Handling a control character in a record" for information regarding single control characters.

Treatment of Unicode BOM

When a line sequential file in a 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.



- 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.

Note

An identification code that is called a BOM at the head of the file is added to the text file of Unicode made with Windows. Specify "CHECK" or "AUTO" for environment variable CBR_FILE_BOM_READ when you use this file with Windows.

6.4 Using Relative Files

This section explains the following:

- Defining relative files.
- Defining records.
- Processing relative files.

```
 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  relative-record-number].
 DATA DIVISION.
   FILE SECTION.
   FD  file-name
      [RECORD  record-size].
   01  record-name.
         Record description entries
   WORKING-STORAGE SECTION.
[01  relative-record-number    PIC 9(5) BINARY.]
 PROCEDURE DIVISION.
      OPEN open-mode  file-name.
     [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 from 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.

The following diagrams uses examples of referencing records to show the process differences between sequential and random access modes:

Figure 6.2 Sequential Access

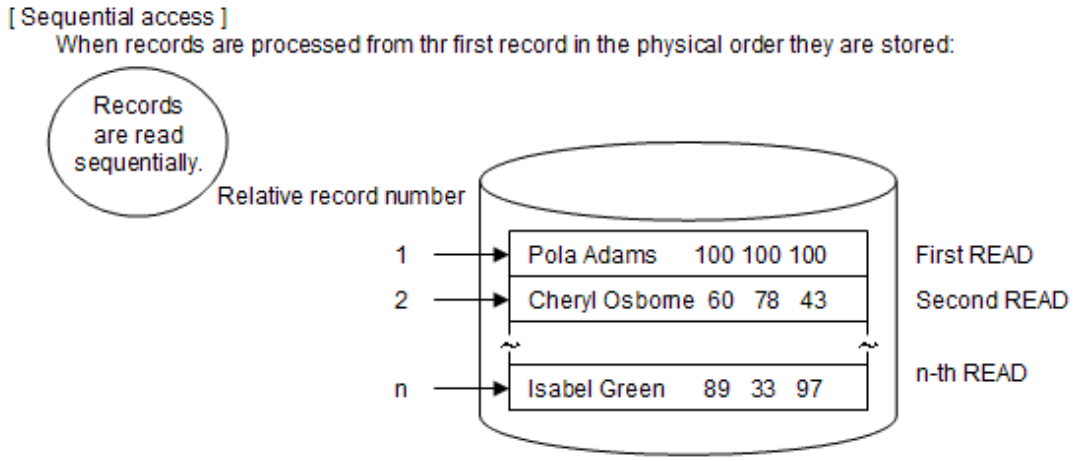
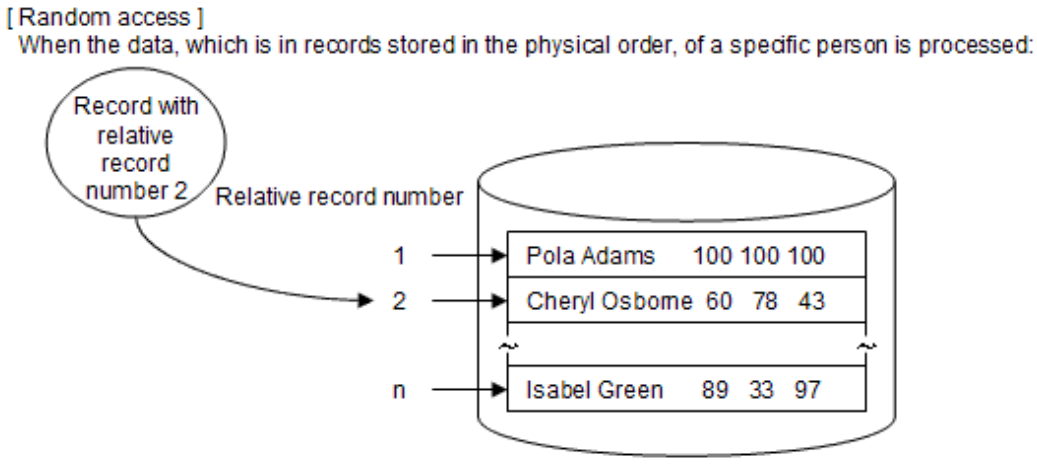


Figure 6.3 Random Access



Relative Record Numbers

Specify the name of the data item in which the relative record number is stored in the RELATIVE KEY clause. In 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.

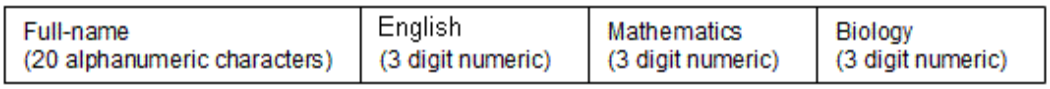
Record Formats and Lengths

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.

An example of relative record definitions in the fixed length record format




```
FD   class-file.
01   grade-record.
      02  Full-name          PIC X(20).
      02  English            PIC 9(3).
      02  Mathematics        PIC 9(3).
      02  Biology            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, execute the OPEN statement in the OUTPUT mode (OUTPUT phrase).

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

Creation (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.

Perform the following procedure to create files:

```
OPEN OUTPUT  file-name.
 Editing-records
 [Setting-relative-record-number]
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the OUTPUT phrase in an OPEN statement.
2. Set the record contents to be written.
3. If in random or dynamic access, set a relative record number in data items specified in the RELATIVE KEY clause.
4. Write the record with a WRITE statement.
5. To write all records, repeat steps 2 to 4, then close the file with a CLOSE statement.

If an attempt is made to create a file that already exists, the file is recreated and the original file is overwritten.

Extension (Sequential)

To extend a relative file, open a file in EXTEND mode, and then write records. In this case, records are added after 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.

Perform the following process to extend files:

```
OPEN EXTEND  file-name.
 Editing-records
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the EXTEND phrase in an OPEN statement.
2. Set the record contents to be written.
3. Write the record with a WRITE statement.
4. To write all the records, repeat steps 2 and 3, then close the file with a CLOSE statement.

Reference (Sequential, Random, and Dynamic)

To refer to records, open the file in INPUT mode, then read file records. In sequential access, specify the start position of the record to be read with a START statement, and 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.

In sequential access

```
OPEN INPUT  file-name.
 [Setting-relative- record number
START  file-name ... . ]
READ   file-name [NEXT] ... .
CLOSE  file-name.
```

1. Open a file with the INPUT phrase in an OPEN statement.
2. If attempting to start the process from a specific record other than the first record, set the relative record number from which you want to start for data items specified in RELATIVE KEY clause. Then execute a START statement.
3. Read the records in ascending order of relative record numbers starting with the first record with a READ statement. If in dynamic access, describe the NEXT phrase in the READ statement.
4. To read all the necessary records, repeat step 3, then close the file with a CLOSE statement.

In random access

```
OPEN INPUT  file-name.
 Setting-relative-record-number
READ   file-name  ... .
CLOSE  file-name.
```

1. Open a file with the INPUT phrase in an OPEN statement.
2. Set the relative record number referred for data items specified in RELATIVE KEY clause.
3. Read the records that have the relative record number set in step 2 with a READ statement.
4. To read all the necessary records, repeat steps 2 and 3, and then close the file with a CLOSE statement.

- If 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, and then execute the REWRITE statement.

In sequential access

```
OPEN I-O  file-name.
 [Setting-relative-record-number
START  file-name  ... .]
READ   file-name [NEXT] ... .
 Editing-records
REWRITE  record-name  ... .]
CLOSE    file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. If attempting to start the process from a specific record other than the first record, set a relative record number you want to start for data items specified in RELATIVE KEY clause. Then execute a START statement.
3. Read the records in ascending order of relative record numbers with a READ statement. If in dynamic access, describe the NEXT phrase in the READ statement.
4. Update the read records.
5. Write updated records in a REWRITE statement.
6. To update all the necessary records, repeat steps 3 to 5, then close the file with a CLOSE statement.

In random access

```
OPEN I-O  file-name.
 Setting-relative-record-number
 [READ   file-name  ... .]
 Editing-records
REWRITE  record-name  ... .
CLOSE    file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.

2. Set a relative record number updated for data items specified in RELATIVE KEY clause.
3. If necessary, read the records that have the relative record number set in step 2 with a READ statement.
4. Update or edit the records.
5. Write updated or edited records in a REWRITE statement.
6. To update all the necessary records, repeat steps 2 to 5, then close the file with a CLOSE statement.

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, and 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, and then execute the DELETE statement.

In sequential access

```
OPEN I-O  file-name.
 [Setting-relative-record-number
START  file-name  ... .]
READ   file-name [NEXT] ... .]
DELETE file-name  ... .
CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. If attempting to start the process from a specific record other than the first record, set a relative record number you want to start for data items specified in RELATIVE KEY clause. Then execute a START statement.
3. Read the records in ascending order of relative record numbers with a READ statement. If in dynamic access, describe the NEXT phrase in the READ statement.
4. Delete the read records with a DELETE statement.
5. To delete all the unnecessary records, repeat steps 3 and 4, then close the file with a CLOSE statement.

In random access

```
OPEN I-O  file-name.
 Setting-relative-record-number
DELETE file-name  ... .
CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Set a relative record number deleted for data items specified in RELATIVE KEY clause. Then execute a START statement.
3. Delete the records with a DELETE statement.
4. To delete all the unnecessary records, repeat steps 2 and 3, then close the file with a CLOSE statement.

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.

```
    OPEN I-O file-name.
     Editing-records
     Setting-relative-record-number
    WRITE  record-name  ... .
    CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Set record contents to be inserted.
3. Set a relative record number inserted for data items specified in RELATIVE KEY clause.
4. Insert the records with a WRITE statement.
5. To insert all records, repeat steps 2 to 4, then close the file with a CLOSE statement.

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

Set relative record numbers for data items in specified in the RELATIVE KEY clause.

For example:

```
MOVE  1  TO  elative-record-number.
```

6.5 Using Indexed Files

This section explains following:

- Defining indexed files.
- Defining records.
- Processing 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 from 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.

The following diagrams show process differences between sequential and random access modes using examples of referencing records:

Figure 6.4 Sequential Access

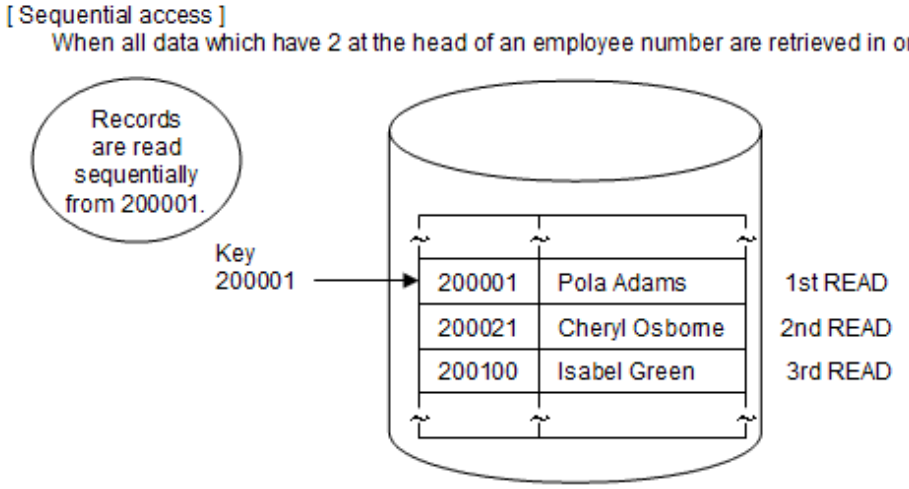
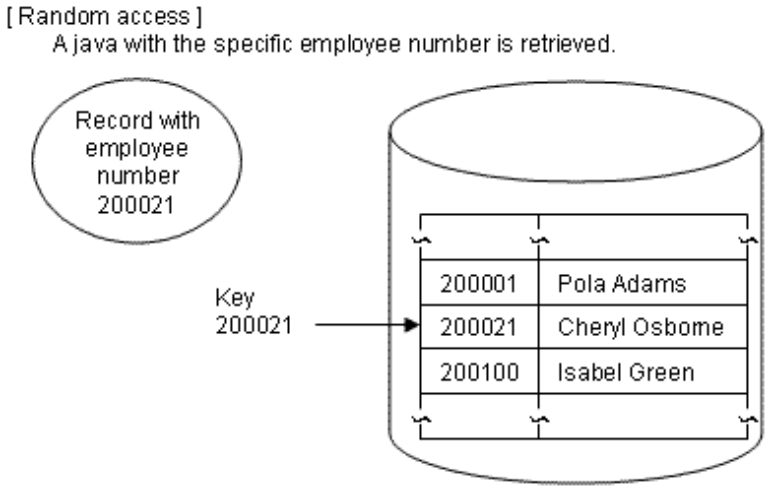


Figure 6.5 Random Access



Prime and Alternate Keys

Keys are classified as prime record keys (prime 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 prime keys, and specific records can be selected with the prime key values. When defining an indexed file, specify the name of a data item as the prime key in the RECORD KEY clause.

As with the prime key, an alternate key can be 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 prime 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

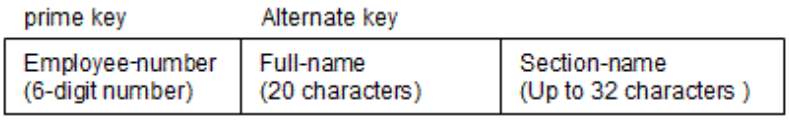
The explanations of the record formats and lengths of indexed files are the same as for record sequential files. See "6.2.2 Defining Record Sequential File Records".

Record Construction

Define the attributes, positions, and sizes of keys and data other than keys in records in the record description entries. Note the following points about defining keys:

- To process an existing file, the number of items, item positions, and sizes to be specified for the prime or alternate keys must be equal to those defined at file creation. The specification order and the number of prime keys must be the same as for the alternate keys.
- If writing two or more record description entries for a file, write the prime key data item in only one of these record description entries. The character position and size defining the prime 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).

An example of record definitions in the variable length records format




```
*>       :
     RECORD KEY IS  employee-number
     ALTERNATE RECORD KEY IS  full-name.
*>       :
DATA DIVISION.
 FILE SECTION.
 FD  employee-file
    RECORD IS VARYING IN SIZE FROM 28 TO 58 CHARACTERS
                               DEPENDING ON record-length.
*>       :
 01  employee-record.
    02  Employee-number  PIC 9(6).
    02  Full-name        PIC X(20).
    02  Section-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 of 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 phrase).

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

Creation (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.

Perform the following procedure to create files:

```
OPEN OUTPUT  file-name.
 Editing-records
 Setting-prime-key-value
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the OUTPUT phrase in an OPEN statement.
2. Set the record contents number to be written.
3. Set prime key values for data items specified in the RECORD KEY clause. In sequential access, write the records in ascending order of prime key values with a WRITE statement.
4. Write the records with the WRITE statement.
5. To read all the necessary records, repeat steps 2 to 4, then close the file with a CLOSE statement.

Note

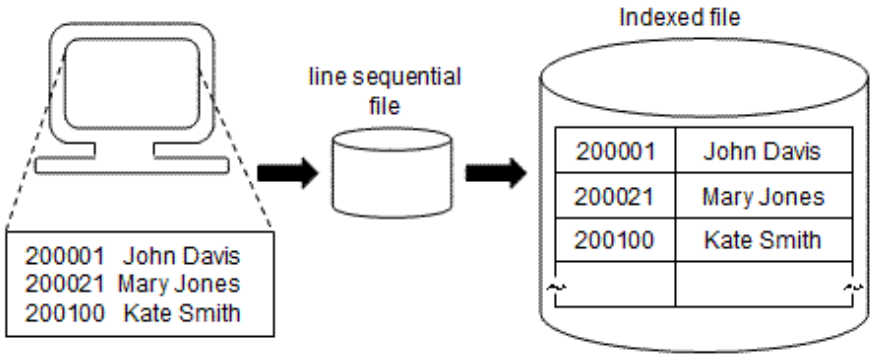
- If an attempt is made to create a file that already exists, the file is recreated and the original file is overwritten.
- Before writing a record, the prime key value must be set. In sequential access, records must be written so that the prime keys are in ascending order.

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, then write it to an indexed file, as shown in the following example:

Figure 6.6 Creating an Indexed File



Extension (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.

Perform the following procedure to extend files:

```
OPEN EXTEND  file-name.
 Editing-records
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the EXTEND phrase in an OPEN statement.
2. Set the record contents to be written.

3. Write the record with a WRITE statement.
4. To write all the records, repeat steps 2 and 3, then close the file with a CLOSE statement.

To edit the records to be written, the prime key values must be in ascending order. Also the first prime key value to be processed must meet the following conditions:

- If DUPLICATES phrase is specified in the RECORD KEY clause in the file control entry:

  (the first prime key value to be processed) >= (the maximum prime key value in the file)

- If DUPLICATES phrase is not specified in the RECORD KEY clause in the file control entry:

  (the first prime key value to be processed) > (the maximum prime key value in the file)

Reference (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 prime or alternate key values. In random access, the records to be read are determined by the prime or alternate key values.

In sequential access

```
OPEN INPUT  file-name.
 Setting-key-value
START  file-name.
READ   file-name [NEXT] ... .
CLOSE  file-name.
```

1. Open a file with the INPUT phrase in an OPEN statement.
2. Set prime key values and alternate key values of the records to be referred to start.
3. Specify the start position of the record to be referred to start with a START statement.
4. Read the records in ascending order of key values specified with a READ statement. If in dynamic access, describe the NEXT phrase in the READ statement.
5. To read all the necessary records, repeat step 4, then close the file with a CLOSE statement.

In random access

```
OPEN INPUT  file-name.
 Setting-key-value
READ   file-name ... .
CLOSE  file-name.
```

1. Open a file with the INPUT phrase in an OPEN statement.
2. Set prime key values and alternate key values of the records to be referred to start.
3. Read the records with a READ statement.
4. To read all the necessary records, repeat steps 2 and 3, then close the file with a CLOSE statement.

- If 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, "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 prime key records with the specified key values are updated.

In sequential access

```
OPEN I-O  file-name.
 Setting-key-value
START  file-name.
READ   file-name [NEXT] ... .
 Editing-records
REWRITE  record-name.
CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Set prime key values and alternate key values of the records to be updated to start.
3. Specify the start position of the record to be updated to start with a START statement.
4. Read the records in ascending order of key values specified with a READ statement. If in dynamic access, describe the NEXT phrase in the READ statement.
5. Update the read records.
6. Write the updated records with a REWRITE statement.
7. To read all the necessary records, repeat steps 4 to 6, then close the file with a CLOSE statement.

In random access

```
OPEN I-O  file-name.
 Setting-key-value
 [READ   file-name ... .]
 Editing-records
REWRITE  record-name.
CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Set prime key values and alternate key values of the records to be updated to start.
3. If necessary, read the records with a READ statement.
4. Edit or update the records.
5. Write the edited or updated records with a REWRITE statement.
6. To update all the necessary records, repeat steps 2 to 5, then close the file with a CLOSE statement.

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 prime 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, and then delete records from the file. In sequential access, records read with the last READ statement are deleted. In random access, the prime key records with the specified key values are deleted.

In sequential access

```
OPEN I-O  file-name.
 Setting-key-value
START  file-name.
READ   file-name [NEXT] ... .
DELETE file-name ... .
CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Set prime key values and alternate key values of the records to be deleted to start.
3. Specify the start position of the record to be deleted to start with a START statement.
4. Read the records in ascending order of key values specified with a READ statement. If in dynamic access, describe NEXT phrase in the READ statement.
5. Delete the read records with a DELETE statement.
6. To delete all the unnecessary records, repeat steps 3 to 5, then close the file with a CLOSE statement.

In random/dynamic access

```
OPEN I-O   file-name.
 Setting-key-value
DELETE file-name ... .
CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Set prime key values and alternate key values of the records to be deleted to start.
3. Delete the records with DELETE statement.
4. To delete all the unnecessary records, repeat steps 2 and 3, and then close the file with CLOSE statement.

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 prime key value.

```
OPEN I-O  file-name
 Editing-records
 Setting-key-value
WRITE  record-name  ... .
CLOSE  file-name.
```

1. Open a file with the I-O phrase in an OPEN statement.
2. Set record contents to be inserted.
3. Set prime key values and alternate key values of the records to be inserted.
4. Write the records with a WRITE statement.
5. To insert all the records, repeat steps 2 to 4, then close the file with a CLOSE statement.

Note

If the records specifying the key values below already exist, the invalid key condition is satisfied. For information about the invalid key condition, see "6.6 Input-Output Error Processing".

- If DUPLICATES phrase is not specified in the RECORD KEY clause in the file control entry.
- If DUPLICATES phrase is not specified in the ALTERNATE RECORD KEY clause in the file control entry.

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 phrase (detecting occurrence of the at end condition).
- INVALID KEY phrase (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 Phrase

When all records in a file are read sequentially and no next logical record exists, file access is completed. This is called an AT END condition. To detect occurrence of the at end condition, specify the AT END phrase in a READ statement. With AT END phrase, processing to be done at occurrence of the at end condition can be written. The following is an example of the coding of a READ statement with AT END phrase:

```
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 Phrase

In input-output processing with an indexed or relative file, an input-output error occurs if the record with a specified key or relative record number does not exist in the file. This is called an invalid key condition. To detect occurrence of the invalid key condition, specify INVALID KEY phrase in the READ, WRITE, REWRITE, START, and DELETE statements. With INVALID KEY phrase, 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 phrase is:

```
MOVE  "Z" TO   prime-key.
READ  indexed-file INVALID KEY
  GO TO invalid-key-processing
END-READ.
```

If no record with prime 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 no I-O status value is checked after the input-output statement, program execution continues even if an input-output error occurs. Subsequent operation is unpredictable.

For I-O status values to be posted, refer to "Appendix B I-O Status List".

Classification of successful I-O status contains any information about the I-O result even if a message saying that the I-O statements are executed successfully.

The following is an example of coding a FILE STATUS clause:

```
    SELECT   file-1
      FILE STATUS IS   input-output-status
*>       :
 DATA DIVISION.
 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 can specify error procedures by writing a USE AFTER ERROR/EXCEPTION statement in Declaratives in the PROCEDURE DIVISION. Writing error procedures executes the processing written 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 with 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 phrase is specified in the READ statement with which the at end condition occurred.
- INVALID KEY phrase 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).

Branch from error procedures to 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.

The following is an example of a description for error procedures:

```
 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               *> [2]
       THEN GO TO   open-error-processing.
*>     :
```

If a file could not be opened, error procedures (the MOVE statement of [1]) are executed and control is passed to the statement (the 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 phrase, INVALID KEY phrase, FILE STATUS clause, and error procedures are written. The following table lists execution results when input-output errors occur. Input-output errors in this section indicate that conditions occurred that contain other than classification of a successful I-O status.

Table 6.4 Execution results when input-output errors occur

<table>
<tr><th colspan="2" rowspan="2">Error type</th><th colspan="2">With error procedures</th><th colspan="2">Without error procedures</th></tr>
<tr><th>With the FILE STATUS clause</th><th>Without the FILE STATUS clause</th><th>With the FILE STATUS clause</th><th>Without the FILE STATUS clause</th></tr>
<tr><td rowspan="2">AT END condition or invalid key condition</td><td>Detected by using an executed input-output statement with AT END or INVALID KEY phrase</td><td colspan="4">The statement written for the AT END or INVALID KEY phrase is executed.</td></tr>
<tr><td>Detected by using an executed input-output statement without AT END or INVALID KEY phrase</td><td>After error procedures are executed, the statement immediately following an input-output statement in which an error occurred is executed.</td><td>The statement immediately following an input-output statement (in which an error occurred) is executed.</td><td>A U-level message is output and the program terminates abnormally.</td></tr>
<tr><td colspan="2">Other input-output errors</td><td>After error procedures are executed, the statement immediately following an input-output statement in which an error occurred is executed.</td><td>An I-level message is output then the statement immediately following an input-output statement (in which an error occurred) is executed.</td><td>A U-level message is output and the program terminates abnormally.</td></tr>
</table>

Information

- If an effective error handling procedure is available and the CBR_FILE_USE_MESSAGE=YES environment variable is specified, I-level messages are output.

```
$ CBR_FILE_USE_MESSAGE=YES ; export CBR_FILE_USE_MESSAGE
```

- To confirm the environment variable set at application execution time, specify CBR_CBRINFO=ENV when the input/output error shows an environment variable error. For details, refer to "Appendix E Environment Variable List".

6.7 File Processing Execution

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 relationship between the contents of the ASSIGN clause and files is explained below.

Using a File Identifier in the ASSIGN Clause

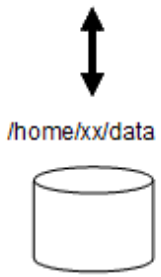
Map the file identifier to the real file name by setting the file identifier as a runtime environment variable.

- Input command

```
$  OUTDATA=/home/xx/data ; export OUTDATA
$  A
```

- Contents of the program

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. A.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
    FILE-CONTROL.
      SELECT file1
            ASSIGN TO OUTDATA.
DATA DIVISION.
  FILE SECTION.
  FD file1.
  01  record1   PIC X(80).
PROCEDURE DIVISION.
    OPEN OUTPUT file1.
    WRITE record1.
    EXIT PROGRAM.
END  PROGRAM A.
```



Executing program A processes the file/home/xx/data:

Note

- A compiler error occurs if a lower-case file identifier is specified in the ASSIGN clause and a file is compiled with compiler option NOALPHAL specified.
- If runtime environment variable information 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.

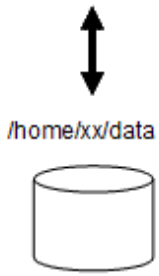
- Input command

```
$  B
```

- Contents of the program

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT file1
      ASSIGN TO "/home/xx/data".
DATA DIVISION.
```

```
 FILE SECTION.
 FD file1.
 01  record1  PIC X(80).
PROCEDURE DIVISION.
    OPEN OUTPUT file1.
    WRITE record1.
    CLOSE file1.
    EXIT PROGRAM.
END  PROGRAM B.
```



Executing program B processes the file/home/xx/data:

Note

When the file name written in the program is a relative path name, the file having the current directory name prefixed is eligible for input-output processing.

Using a Data Name in the ASSIGN Clause

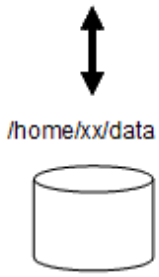
Input-output processing uses the file name specified in the data item.

- Input command

```
$  C
```

- Contents of the program

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. C.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT file1
      ASSIGN TO dataname1.
DATA DIVISION.
 FILE SECTION.
 FD file1.
 01  record1     PIC X(80).
 WORKING-STORAGE SECTION.
 01  dataname1  PIC X(30).
PROCEDURE DIVISION.
    MOVE "/home/xx/data" TO dataname1.
    OPEN OUTPUT file1.
    WRITE record1.
    CLOSE file1.
    EXIT PROGRAM.
END  PROGRAM C.
```



Executing program C processes the file/home/xx/data:

Note

- If the file name specified in the program is a relative path name, the file having the current directory name prefixed is eligible for input-output processing.
- If the data name is left blank, a file assignment error occurs.

Using "DISK" in the ASSIGN Clause

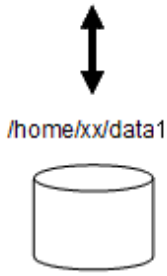
Input-output processing uses the file name prefixed by the current directory name specified in the SELECT clause.

- Input command

```
$  cd /home/xx
$  D
```

- Contents of the program

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. D.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT data1
      ASSIGN TO DISK.
DATA DIVISION.
 FILE SECTION.
 FD Data1.
 01  record1     PIC X(80).
PROCEDURE DIVISION.
    OPEN OUTPUT Data1.
    WRITE record1.
    CLOSE Data1.
    EXIT PROGRAM.
END  PROGRAM D.
```



Executing program 'D' processes the file/home/xx/data1:

Note

For SELECT clause, file names of absolute file names cannot be specified (because the rules for user-defined words used in COBOL disallow it).

6.7.2 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 regenerated and the original contents are overwritten.

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, 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

- If a program is terminated without executing a CLOSE statement, files are closed unconditionally. This is called unconditional close. 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 generated, the files become unusable, and they remain open.
- A file can be assigned with the file identifier, in which case an input-output statement is executed for the file assigned with the file identifier, even if the file assignment destination is changed with the environment variable operation function while the file is open. If 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 operation 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 unpredictable. 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 unpredictable.
- If 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 may be recovered with the Recovery command of the COBOL File Utility. For more details, refer to "Chapter 19 COBOL File Utility".

6.7.3 Locking Control of Files

Disable access during file processing by setting the locking mode for files or locking records in use. This is called locking control of files.

Locking control when accessing files is guaranteed only when using COBOL programs, COBOL file access routines, and the COBOL file utility. If different languages and tools are used to access those files, locking control is not guaranteed.

This section explains the relationship between file processing and locking control of files.

6.7.3.1 Setting Files in Exclusive Mode

If 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 in the OUTPUT mode is executed.

The following tables show the file sharing mode conditions for the possible combinations of LOCK MODE, OPEN with or without the WITH LOCK phrase, and the OPEN statement modes:

Table 6.5 When LOCK MODE phrase is not specified

OPEN statement MODE	INPUT		I-O		EXTEND		OUTPUT	
WITH LOCK phrase in an OPEN statement	Without	With	Without	With	Without	With	Without	With
locking mode	Shared	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive

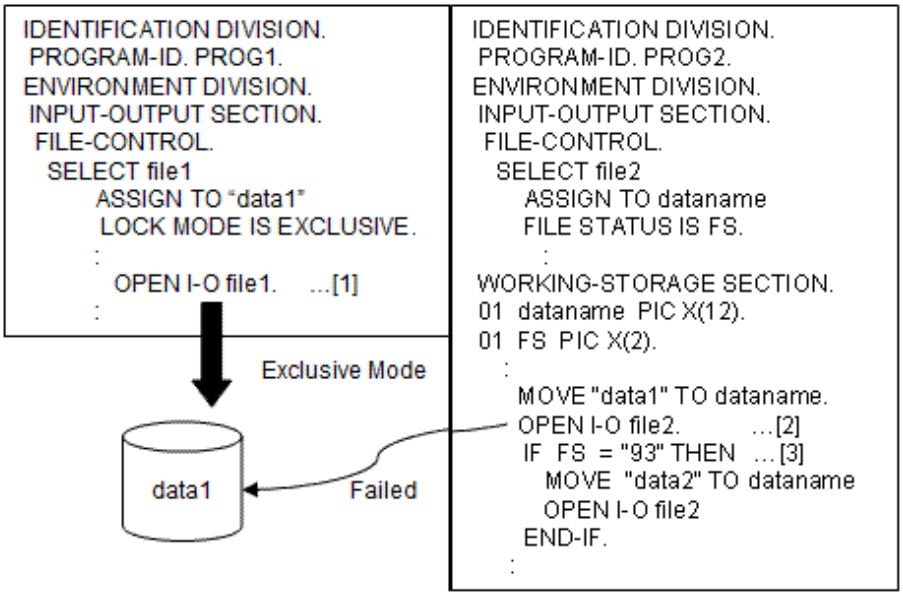
Table 6.6 When EXCLUSIVE phrase is specified in the LOCK MODE clause

OPEN statement MODE	INPUT		I-O		EXTEND		OUTPUT	
WITH LOCK phrase in an OPEN statement	Without	With	Without	With	Without	With	Without	With
locking mode	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive	Exclusive

Table 6.7 When AUTOMATIC or MANUAL phrase is specified in the LOCK MODE clause

OPEN statement MODE	INPUT		I-O		EXTEND		OUTPUT	
WITH LOCK phrase in an OPEN statement	Without	With	Without	With	Without	With	Without	With
locking mode	Shared	Exclusive	Shared	Exclusive	Shared	Exclusive	Exclusive	Exclusive

Figure 6.7 Exclusive Mode



Explanation of figure

- [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.

6.7.3.2 Locking Records

If records are locked, other users cannot process them (however, reference process can be executed). To lock only records in use, first 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 in 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 fails. Records in a file opened in share mode are locked with the execution of I-O statements 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.

The following table shows record status with the above combinations:

Table 6.8 Record status (excessive/Shared)

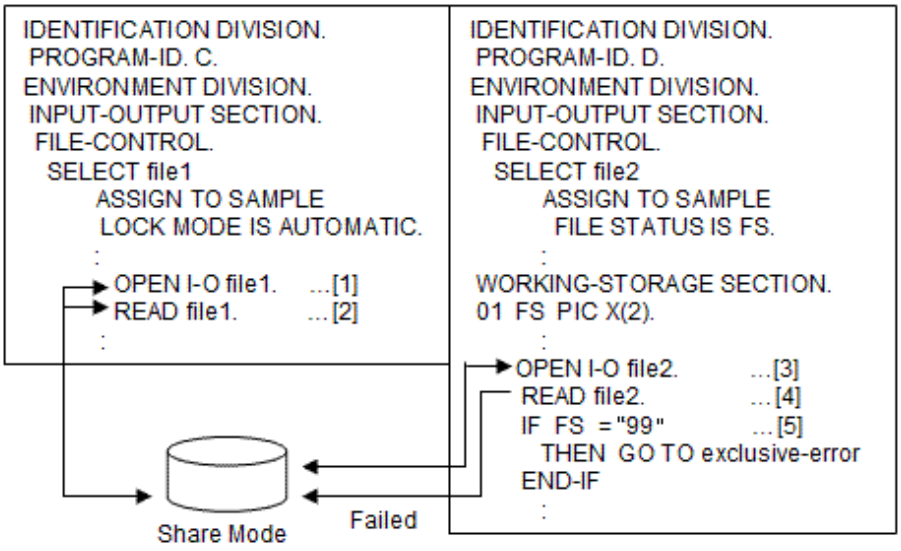
LOCK MODE clause description	AUTOMATIC			MANUAL		
READ WITH LOCK phrase	No phrase	WITH LOCK	WITH NO LOCK	No phrase	WITH LOCK	WITH NO LOCK
Record locked	Yes	Yes	No	No	Yes	No

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:

Figure 6.8 Record Locking



Explanation of figure

- [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.

6.8 Other File Functions

This product supports functions to connect to other file systems and to improve performance. This section explains the following functions:

- Process for improving performance
- Additional writing of a file
- Connecting files
- Dummy File
- Named pipe
- External File Handler

The following table shows the I-O function ranges used by the COBOL file system. The range of I-O functions that can be used in the external file handler depends on the file system used.

Table 6.9 Functional differences between file systems

Classification	Items		COBOL File System
File	Maximum size of files (bytes)	Record sequential files	To system limit (*1)
		Record sequential files (BSAM specification)	To system limit (*1)
		Line sequential files	To system limit (*1)
		Line sequential files(BSAM specification)	To system limit (*1)
		Relative files	To system limit (*1)
		Indexed files	To system limit (*1)
	Maximum count of concurrent open of same file		1,024
Record	Record formats	Fixed length record	o
		Variable length record	o
	Maximum length of records(bytes)	Fixed length record	32,760
		Variable length record	32,760
	Minimum length of records(bytes)	Record sequential files	1
		Line sequential files	0
		Relative files	1
		Indexed files	Size to key
File control entry	SELECT clause	OPTIONAL phrase	o
	ASSIGN clause	File-Reference-Identifier specification	o
		File-identifier-literal specification	o
		Data-name specification	o
		DISK specification	o
	FILE STATUS clause	File status	o
	LOCK MODE clause	AUTOMATIC	o
		EXCLUSIVE	o
		MANUAL	o
	RECORD KEY clause	Maximum numbers of data that can be specified	254
		Maximum data length that can be specified (bytes)	254
	ALTERNATE RECORDKEY clause	Maximum numbers of data that can be specified	254 (*2)
		data length that can be specified (bytes)	254
	RELATIVE KEY phrase	Maximum value of specifiable data items	9,223,372,036,854,775,807
	Record key items(*3)	Alphanumeric	o
		National language	o
		Unsigned zoned decimal	o
		Signed zoned decimal	-
		Unsigned packed-decimal	o

Classification	Items		COBOL File System
		Signed packed-decimal	-
		Unsigned binary(BINARY)	o
		Unsigned binary(COMP-5)	-
		Signed binary(BINARY/COMP-5)	-
Statement	READ statement	WITH LOCK phrase	o
	START statement	Full key	o
		Partial key	o
	DELETE statement	Deletion of duplicated key values	o
	UNLOCK statement		o
Function	Thread	Process (single thread)	o
		Multithread	o
	Transaction control	Display of transaction start	-
		COMMIT order	-
		ROLLBACK order	-
	Recovery		o (*4)

o : Supported

- : Not supported

*1 : The maximum file size for COBOL in another UNIX system is the system limit size, regardless of whether a required high capacity file is specified. If a high capacity file is specified, the system operates as usual.

*2 : 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. Therefore, if the amount of data specified in the RECORD KEY clause is larger, this value becomes smaller.

*3 : If unsupported data items are defined in record keys, the execution results are undefined.

*4 : In COBOL File System, the function to recover index file that became inaccessible is being offered.

6.8.1 Process to Improve Performance

Buffering and high-speed access processes are provided to improve performances for file access. Note that each function of these processes has some restrictions.

6.8.1.1 Improving Performances for File Processing

I-O process performances for sequential, line sequential, or relative files opened with input mode can be improved. This section explains how to use this function and things to note about this function.

Using this function

Set "yes" as the value for the environment variable CBR_INPUT_BUFFERING, as in the following example:

```
$  CBR_INPUT_BUFFERING=yes ; export CBR_INPUT_BUFFERING
```

Note

To reduce the number of times that COBOL runtime system accesses files, this function is executed by units of buffers rather than units of records when processes read records. A Single buffer may contain multiple records. Therefore, if file contents are updated by other file-identifiers, the latest record contents may be guaranteed. This specification is not available for "6.8.1.2 High-speed File Processing".

6.8.1.2 High-speed File Processing

Access performances can be speed up by restricting the ranges used for record sequential files and line sequential files.

High speed file processing can be used in the following cases:

- A file is opened in exclusive mode, and is written to as an output file.
- An input restricted file is read.

This section explains how to specify this function and things to note about executing high-speed file processing.

Specification Methods

The methods of specification for sequential files and line sequential files are the same.

- When defining a data-name as a file-reference-identifier in the program

  Specify ",BSAM" following the file-name to be allocated upon setting of environment variable information. For details on setting environment variables information, refer to "6.7.1 Assigning Files" in this chapter.

```
$  file-identifier=file-name,BSAM ; export file-identifier
```

- "ipf,x64"When defining a file-identifier literal as a file-reference-identifier in the program

  Specify ",BSAM" following the file-name to be allocated at definition of the file-identifier literal in the program:

```
  ASSIGN TO "file-name,BSAM"
```

- When defining a data name as a file-reference-identifier in the program

  Specify ",BSAM" following the file-name to be allocated at definition of the data-name in the program:

```
  MOVE  "file-name,BSAM" TO data-name
```

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 phrase. Operation is not guaranteed if a file is opened without specifying INPUT phrase. File sharing is not permitted in the same process. 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, attempting to access a file from other programs is likely to result in an open error.
- 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 large capacity file function cannot change the maximum file size. Usually, the maximum size cannot exceed the normal system limit. Note that normal processing can continue even if the large capacity file function is specified.
- 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.

Batch specification

This section describes the batch specification for "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 ; export CBR_FILE_SEQUENTIAL_ACCESS
```

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
Connecting files (*)	CONCAT
Dummy File	DUMMY
Other File System	EXFH
	(Named Pipe)

* : This function can be used for the file when high-speed processing is also enabled for that file

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 has these limitations. Refer to "Notes on shared files" for more information.

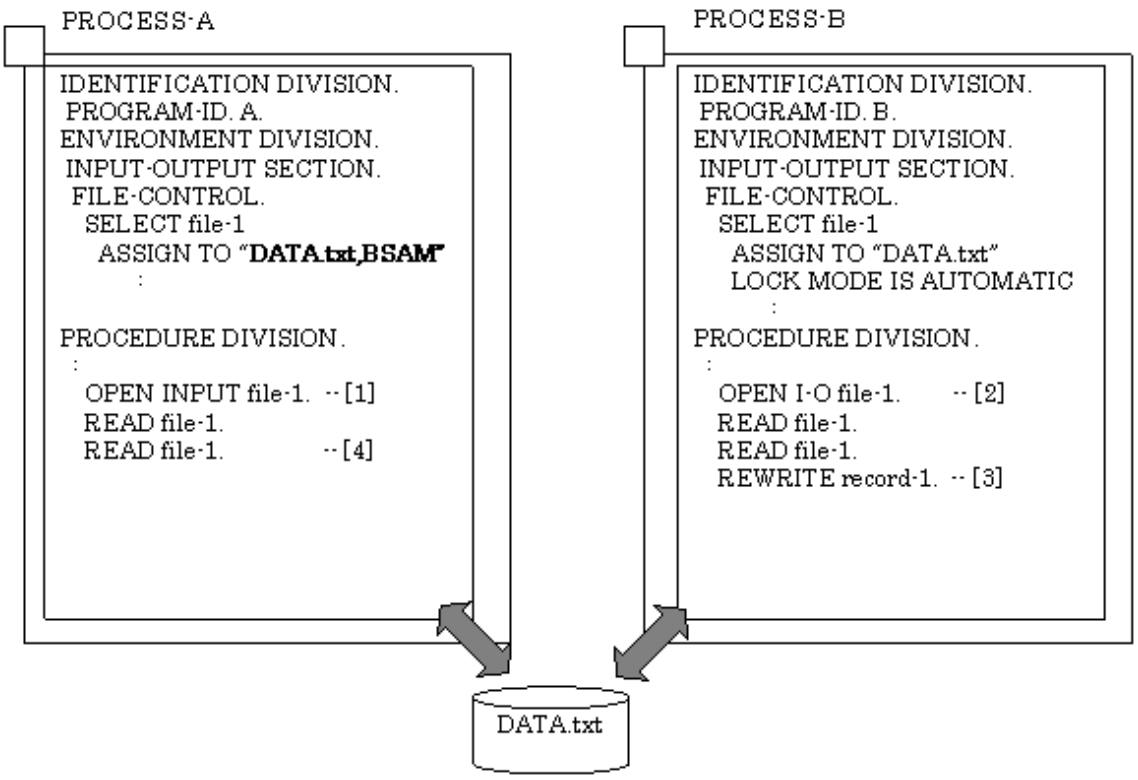
Notes on shared files

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.



[1] PROCESS-A : The file is opened in shared mode with INPUT specified. (High-speed File Processing)

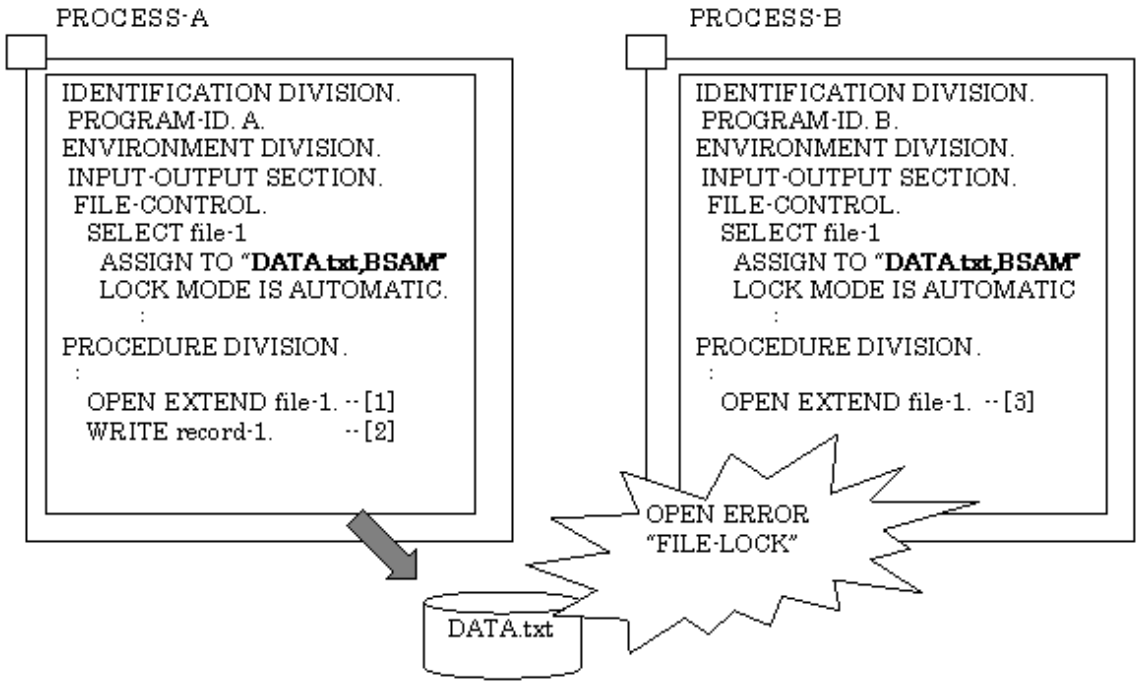
[2] PROCESS-B : The file is opened in shared mode with I-O specified.

[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.



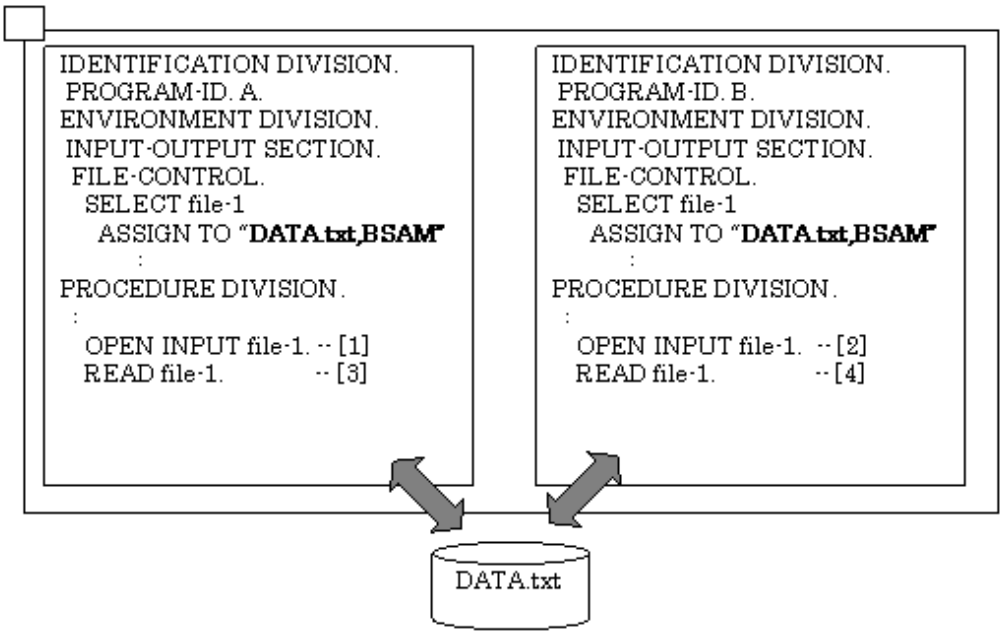
[1] PROCESS-A : The file is opened in shared mode with EXTEND specified.

[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.



[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.

6.8.2 Process Related Writing Files

This section explains the functions provided to write files.

6.8.2.1 Immediate Reflection for Written Contents When Closing

Written contents can be reflected immediately a CLOSE statement is executed. This section explains how to use this function and things to note about this function.

Using this function

Set "yes" as the value for the environment variable CBR_CLOSE_SYNC, as in the following example:

```
$  CBR_CLOSE_SYNC=yes ; export CBR_CLOSE_SYNC
```

Note

This function issues an order to write buffer contents managed by the OS when a CLOSE statement is executed in disks. Therefore, the performance may deteriorate, depending on whether the OS buffer condition is using this function.

6.8.2.2 Specification of Trailing Blanks in the Line Sequential File

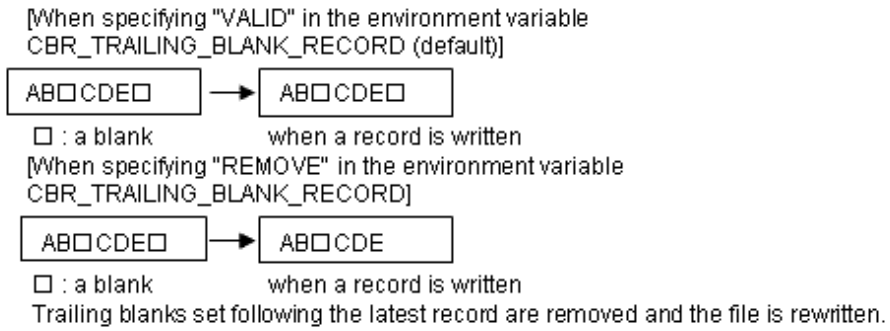
Whether or not to delete trailing blanks in the line sequential files when a WRITE statement is executed can be specified. This section explains how to use this function.

Using this function

If "REMOVE" is the value set for the environment variable CBR_TRAILING_BLANK_RECORD, trailing blanks in a record are removed when a WRITE statement is executed.

If "VALID" is the value set for the environment variable CBR_TRAILING_BLANK_RECORD, trailing blanks in a record are not removed. If value is omitted, "VALID" is assumed to be specified.

```
$  CBR_TRAILING_BLANK_RECORD={ REMOVE | VALID } ; export CBR_TRAILING_BLANK_RECORD
```



Example

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.

6.8.3 COBOL File Access Routine

The NetCOBOL File Access Subroutines consists of Application Program Interface (API) functions for accessing all types of COBOL file. This routine supports development and operation of applications that access the COBOL files. For details on this routine, refer to the "NetCOBOL File Access Subroutines User's Guide".

6.8.4 Adding records to a file

Records can be added to an existing file by executing the OPEN OUTPUT statement. This section explains how to use this function.

Using this function

Specify ",,MOD" after the file name to be allocated for the file-identifier.

```
$  file-identifier=file-name,,MOD ; export file-identifier
```

Note

- This function is available only for COBOL record sequential files. This function cannot be specified for another file organization.

6.8.5 Connecting files

Multiple files can be connected in order to reference and update records. This section explains how to use this function.

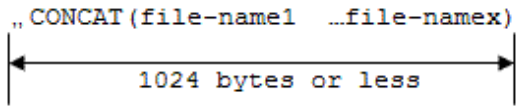
Using this function

Specify ",,CONCAT" for a file identifier.

```
$  file-identifier=",,CONCAT(file-name-1 file-name-2 ... )" ; export file-identifier
```

Note

- Delimit file names by using a blank character.
- If a file name includes a blank character, enclose it between double quotation marks (").
- Only COBOL record sequential files can be processed. This function cannot be specified for another file organization or another file system.
- This function can be specified only for COBOL record sequential files. This function cannot be specified for another file organization.
- If an OPEN statement with OUTPUT or EXTEND specified is executed, an error occurs.
- If the same files are specified, the operation performed is the same as that performed when different files are specified.
- The file-identifier character string cannot exceed 1024 bytes. Therefore, the number of files that can be connected depends on the length of the combined file names. When this function is used for the file-identifier, specify it as follows:



  An error will occur at OPEN sentence execution time if the character string specified in the Connecting files function exceeds 1024 bytes.

6.8.6 Dummy files

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.

Using this function

",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; export file-identifier
```

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.

Functional range

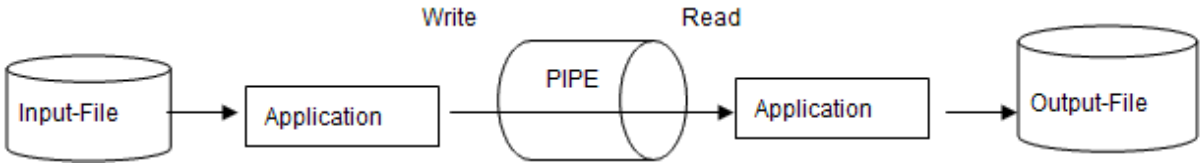
Table 6-10 shows the range where the Dummy file becomes effective.

Table 6.10 Functional range of Dummy file

File Types	Record sequential file Line sequential file Relative file Index file		
Open Mode	OUTPUT EXTEND I-O INPUT		
Input-Output 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.

6.8.7 Named pipe

A named pipe can be used as a sequential file. When data is received and passed between COBOL applications, a named pipe can be used instead of an intermediate file. COBOL applications can be operated in parallel when a named pipe is used.



Using this function

The named pipe is created before the COBOL application is executed. The named pipe is specified similar to allocating a normal file.

Note

The named pipe is made by the mkfifo system command.

Functional range

Table 6.11 Indicates where a named pipe can be used.

File Types	Record sequential files Line sequential files
Record Format	Fixed length format Variable length format

File control entry	SELECT clause	File identifier (*1)
	LOCK MODE clause	Ignored.
Input-Output statement	OPEN statement	OUTPUT/INPUT mode (*2) WITH LOCK specification is ignored.
	READ statement	WITH LOCK/WITH NOLOCK specification is ignored.
	WRITE statement	
	REWRITE statement	Cannot be specified.
	UNLOCK statement	Ignored.

*1: File-identifier literal, data names and DISK cannot be specified.

*2: OPEN mode I-O cannot be specified.

6.8.8 External file handler

The external file handler interface can be used to provide support for COBOL record sequential files, line sequential files, relative files and indexed files. This is provided for compatibility with the equivalent Micro Focus COBOL feature; interface details are not documented.

Using this function

- 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. For details on setting environment variable information, refer to "Assigning Files" in this chapter.

```
$  file-identifier=file-name,EXFH ; export file-identifier
```

- 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
```

Usage

When an external file handler is used, the shared object file name and the entry point 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; export CBR_EXFH_API
```

entry-point-name: The entry point name of the EXFH file system implementation.

```
$  CBR_EXFH_LOAD=shared-object-file-name; export CBR_EXFH_LOAD
```

shared-object-file-name: The shared object file 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 directory.

Note

When CBR_EXFH_LOAD is not specified, "lib*entry-point-name*.so" is treated as a shared object file name.

[Example 1]

When the entry point name of the EXFH file system implementation is "flsys" and the shared object file name is "libfilesys.so", you would specify:

```
$  CBR_EXFH_API=flsys; export CBR_EXFH_API
$  CBR_EXFH_LOAD=libfilesys.so; export CBR_EXFH_LOAD
```

[Example 2]

When the entry point name of the EXFH file system implementation is "file" and the shared object file name is "libfile.so", you can specify:

```
$  CBR_EXFH_API=file; export CBR_EXFH_API
```

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)"; export file-identifier
```

The info-file is a text file with the following content:

```
[EXFH]
CBR_EXFH_API=entry-point-name
CBR_EXFH_LOAD=shared-object-file-name
```

entry-point-name: The entry point name of the EXFH file system implementation.

shared-object-file-name: The shared object file name of the EXFH file system implementation.

Note

When CBR_EXFH_LOAD is not specified, "lib*entry-point-name*.so" is treated as a shared object file name.

Example

When the file name is "Afile", the info-file name is "aflsys.inf", the entry point name is "sflsys" and the shared object file name is "libafilesys.so", you would specify:

```
$  file-identifier="Afile,EXFH,INF(aflsys.inf)"; export file-identifier
```

The content of "aflsys.inf":

```
[EXFH]
CBR_EXFH_API=aflsys
CBR_EXFH_LOAD=libafilesys.so
```

Priority order for determining which external file handler to use

The order of priority for selecting the external file handler is as follows:

- First - the program specified in the external file handler information file
- Second - the program specification in the execution environment variable

Precautions

- The external file handler to be used must be a shared object file (lib*.so). An object file cannot be used, unlike Micro Focus COBOL.

- An external file handler cannot be used from a COBOL application compiled with Unicode.
- When using an external file handler from a COBOL application that was compiled for multithreading, the external file handler must also support multithreading.
- When using an external file handler, FILE STATUS values that are returned from the external file handler are set up as 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.

6.8.9 Cautions

Byte number of character string that can be specified for file-identifier

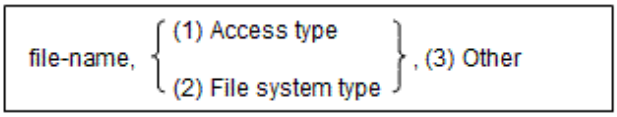
File-identifier character strings must be 1024 bytes or less. If the character string exceeds 1024 bytes, only the first 1024 bytes are processed. An error will occur at OPEN sentence execution time if the file connection function character string exceeds 1024 bytes.

Combining operations

File functions can be classified into 3 types:

(1) Access type	BSAM	High-speed File Processing
	DUMMY	Dummy file
(2) File system type	EXFH	External File Handler
(3) Other	MOD	Appending to existing files
	CONCAT	Connecting files

Specify file functions using the following format. If the specification order is different, the file function will be ignored.



Operations that can be specified at the same time are as follows:

- BSAM in (1) and (3) can be specified at the same time. Both are valid.

  Example 1 BSAM and (3)

```
file-name,BSAM,MOD
,BSAM,CONCAT(file-name-1 file-name-2 ...)
```

- DUMMY in (1) and other functions can be specified at the same time. However, only the Dummy File becomes effective; the other functions are invalid.

  Example 2 BSAM and DUMMY

```
file-name,BSAM,DUMMY
```

  Example 3 (2) and DUMMY

```
file-name,EXFH[,INF(info-file)],DUMMY
```

  Example 4 (3) and DUMMY

```
file-name,,MOD,DUMMY
,,CONCAT(file-name-1 file-name-2 ...),DUMMY
```

  NOTE: DUMMY can be specified immediately after the first comma.

```
,DUMMY,CONCAT(file-name-1 file-name-2 ...)
```

Example 5 BSAM, (3), and DUMMY

```
file-name,BSAM,MOD,DUMMY
,BSAM,CONCAT(file-name-1 file-name-2 ...),DUMMY
```

- DUMMY in (1) and (3) becomes an error when the OPEN statement is executed except when the specification of the ASSIGN clause is the file identifier name.
- When an invalid combination is specified, it is processed as follows:
  - When (2) is specified first, the function specified after (2) becomes invalid, and processing continues.
  - In all other cases, an error occurs when the OPEN statement is executed.

Chapter 7 Printing

NetCOBOL uses a variety of printing methods which are described in this chapter. With NetCOBOL, you can select the printing device, control print characters, and set environment variables for printing. This chapter also explains how to use print files and control records.

7.1 Types of Printing Methods

Use a print 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), print information files, and form descriptors. The print function you use depends on the printer being used.

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.
- Use the print (spool) file only to print. For example, do not use the print (spool) file to input, process, and output.

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 (line by line; How to print data in line mode using), print data by setting print information with an FCB ("7.3 How to Use a Control Record").

By using a print file with a FORMAT clause, functions using the print file without a FORMAT clause and data in forms with form descriptors can be printed.

This chapter classifies print in the following groups:

- [1] Print files without a FORMAT clause ("7.2 How to Print Data in Line Mode Using a Print File").
- [2] Print files without a FORMAT clause ("7.3 How to Use a Control Record").
- [3] Print files with a FORMAT clause.

The following table lists the characteristics, advantages, and uses of printing methods [1] to [3].

Table 7.1 Characteristics, advantages, and uses of each printing method

File Type		[1]	[2]	[3] (*1)
Characteristics	Data can be printed in line mode.	B	B	B
	Data overlaid with FORM overlay patterns can be printed.	C	B (*2)	B
	Data in forms can be printed with form descriptors.	C	C	B
Advantage	Simple program coding.	A	B	B
	Simple forms printing.	B	A	A
	Existing form descriptors generated by other systems can be used.	C	C	A
	Various types of print information can be specified in programs.	C	A	A
Use	Print forms.	B	A	A

A: Can be used (suitable). B: Can be used. C: Cannot be used.

*1: For information about data streams and supported printers that can be specified, refer to the "PowerFORM Runtime Reference".

*2: A FORM overlay pattern in the KOL6 format cannot be output in a PostScript level 1 data stream.

The following table lists related products.

Table 7.2 Related products

File Type		[1]	[2]	[3]
Related Product	PowerFORM	B	A(*1)	A (*2)
	PowerFORM Runtime	B	A(*1)	A
	Interstage List Creator Enterprise Edition (*3)	B	B	A

A: Can be used. B: Cannot be used.

*1: Use to create a FORM overlay pattern.

*2: Use to create a FORM overlay pattern or a form descriptor. For information about notes on using the form descriptors created by PowerFORM, refer to "PowerFORM Runtime Reference".

*3: Use to output form in PDF file.

Outline for each printing method is explained below.

[1] Print File without a FORMAT Clause

A print file is not a particular type of file. It refers to the file defined to be printed.

A print file without a FORMAT clause is defined as the same way as the record sequential files, data can be sent in line mode to a printer with a WRITE statement. 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 a Print File".

[2] Print File without a FORMAT Clause

Control records are output with a WRITE statement. print information, such as FCB, can be specified in a print file.

For details on FCB, see "7.1.6 FCB". For how to use print files, see "7.3 How to Use a Control Record".

[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. However, if printing a form with a print file with a FORMAT clause, PowerFORM RTS must be used.

If the control record specified a FORM overlay pattern is output, a page of output data can be overlaid with the FORM overlay pattern.

For details on FORM overlay patterns, see "7.1.7 Form Overlay Patterns".

For details on form descriptors, see "7.1.8 Form Descriptors".

For how to use a print file with a FORMAT clause, see "7.4 Using Print Files with a FORM Descriptor".

7.1.2 Print Devices

A print file without the FORMAT clause can be used to output PostScript (level 1 and level 2) data streams.

The standard data stream is PostScript level 1.

PowerFORM RTS is required to print forms overlaid with the FORM overlay patterns by PostScript level 1, or print by PostScript level 2.

Note

For information about data stream supported printer devices, refer to manuals of the attached printers.

7.1.3 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.

If printing a national language, the CHARACTER TYPE clause for the national language data item must be specified. The results of printing are not guaranteed if the CHARACTER TYPE clause is omitted when printing national language data items or a national language is printed using alphanumeric data items.

Option	Attribute of print characters					
	Size	Font	Style	Form	Direction	Space
CHARACTER TYPE MODE-n	Available	-	-	Available (*2)	-	Available (*3)
CHARACTER TYPE mnemonic-name	Available	Available	-	Available	Available	Available (3)
CHARACTER TYPE print mode name	Available	Available	Available (*1)	Available	Available	Available

*1: "FONT-nnn" must be specified in FONT of PRINTING MODE clause.

*2: "BY mnemonic-name" must be specified 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 is 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" in the "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" in the "NetCOBOL Language Reference".

The printable character types depend on the functions of each printer.

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:

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 character space set up. If both the character size and space are omitted, the following defaults are used: - National language data item: Printed in 12-point character size - Alphanumeric data item: Printed in 7-point character size

For specification details, see the "NetCOBOL Language Reference".

Note

If the character size specified cannot be used with the printer device, the character size is determined by the printer device specification.

Available Fonts

The following fonts can be specified: 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 If the font is omitted, MINCHOU is used as the default.
Print mode name(if FONT is specified in PRINTING MODE clause).	MINCHOU/MINCHOU-HANKAKU/GOTHIC/GOTHIC-HANKAKU/,GOTHIC-DP/FONT-NUMBER If the font is omitted, the following defaults are used: - National language data item: MINCHOU - Alphanumeric data item: GOTHIC

For the specification details, see the "NetCOBOL Language Reference".

Note

- The FONT-NUMBER in the value specified for "FONT-nnn" in the PRINTING MODE clause.
- If using PostScript for a data stream, the font that can be printed is specified by FONT-NUMBER. In this case, the font face name font that corresponds to the FONT-NUMBER respectively specified in the font table is used and printed. However, in the following cases, the font for the default is used and printed:
  - When no font table name is specified.
  - When the font face name is specified that corresponds to the FONT-NUMBER in a font table.

  For more details about the font table, refer to "7.1.9 Font Tables".
- If a font specified cannot be used with the printer device, the font printed is determined by the printer device specification.

Available Styles

The following styles can be specified: REGULAR, BOLD, ITALIC, and BOLD-ITALIC.

Method

For information about how to specify a character style, refer to "7.1.9 Font Tables".

The default font is REGULAR.

Note

The style can be specified for the font specified by FONT-NUMBER.

Available Print Character Forms

The following print character forms can be specified: em-size and em-size tall and wide; double size, en-size, and en-size tall and wide; double size.

Method

Specification	Character Form
Function name associated with mnemonic-name of MODE-n.	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".

Note

If a print character form specified cannot be used with a printer device, the print character form to be printed is determined by the printer device specification.

Direction of Print Characters

Horizontal or vertical writing can be specified.

Method

Specification	Direction of Print Characters
MODE-1/ MODE-2/ MODE-3	Horizontal writing.
Function name associated with mnemonic-name.	Horizontal or vertical writing. The default direction of print characters is horizontal writing.
Print mode name(if ANGLE phrase is specified in PRINTING MODE clause).	Horizontal or vertical writing. The default direction of print characters is horizontal writing.

For the specification details, see the "NetCOBOL Language Reference".

Note

If the direction of print characters specified cannot be used with the printer device, the direction is determined by the printer device specification.

Print Character Spaces

Character spaces can be specified from 0.01 to 24.00 (unit: cpi).

Method

Specification	Print Character Spaces
MODE-1/ MODE-2/ MODE-3	Spaces determined according to the sizes and forms of print characters. See "Table 7.3 Relationship between print character sizes/forms and character spaces ".
Function name associated with mnemonic-name.	
Print mode name (if PITCH phrase is specified in PRINTING MODE clause).	Character spaces of 0.01 to 24.00 cpi are specified in units of 0.01 cpi. If the specification for character spaces is omitted, character spaces corresponding to the character size specified are used and

Specification	Print Character Spaces
	printed. If the character size is omitted, character spaces are as follows: - National language data items: 6.00 cpi - Alphanumeric data items: 10.00 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							
	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

(Unit: cpi)

See

For the meanings of the codes for character sizes in this table, refer to the "NetCOBOL Language Reference".

Note

If character spaces specified cannot be used with the printer device, the character spaces are determined by the printer device specification.

7.1.4 Setting Environment Variables

The following table shows environment variables that must be set when programs that execute print process are executed:

Table 7.4 Necessary environment variables for program execution

Environment Variable	Value
CBR_LP_OPTION	The lp option to be used.
CBR_PRINTFONTTABLE	Path name for font tables.
CBR_PRT_INF	Path name for print information files.
CBR_FCB_NAME	FCB name for the default value.
FCBDIR	Storage directory for FCB.
FOVLDIR	Storage directory for Form Overlay Patterns.
File-identifiers specified in the ASSIGN clause	Path name for output destinations or printer information files.
PRINTER-n (n=1~9)	Path name for output destination files.
LD_LIBRARY_PATH	Storage directory for subprograms and libraries offered by the system.

CBR_LP_OPTION

If PRINTER is specified for the ASSIGN clause in the file control entry with a print file without FORMAT clause, specify the options that the lp command passes in the CBR_LP_OPTION environment variable.

Example

When specifying the printer device (lp0) and copy (2):

```
$ CBR_LP_OPTION="-dlp0 -n2" ; export CBR_LP_OPTION
```

Note

- The options specified with this function are added after the default options are specified. Therefore, if the same option is specified, an lp command error message may be output.
- This function is not valid for the following files:
  - Print files with FORMAT clause.
- If printing data using this environment variable, the print environment for the lp command must be set in advance.

CBR_PRINTFONTTABLE

Specify CBR_PRINTFONTTABLE when using a common font table for the entire run unit. For information about font tables, refer to "7.1.9 Font Tables".

Example

When specifying the font table file, font table after /home/usr1:

```
$  CBR_PRINTFONTTABLE=/home/usr1/fonttable ; export CBR_PRINTFONTTABLE
```

CBR_PRT_INF

Specify CBR_PRT_INF when using a common print information file for the entire run unit by a print file without FORMAT clause. For information about print information files, refer to "7.1.5 Print Information Files".

Note

When specifying the print information file, insatsu.inf after /home/usr1:

```
$  CBR_PRT_INF=/home/usr1/insatsu.inf ; export CBR_PRT_INF
```

CBR_FCB_NAME

Specify CBR_FCB_NAME when changing the default value for FCB by a print file with FORMAT clause. For information about FCB, refer to "7.1.6 FCB".

Example

When using FCB1 as the default FCB name:

```
$  CBR_FCB_NAME=FCB1 ; export CBR_FCB_NAME
```

Note

- Specify FCB names with up to 4 alphanumeric characters.
- If you specify a default FCB, the FCB storage directory must be specified with the FCBDIR environment variable.
- The default FCB name for print files without a FORMAT clause, is specified by the print information file.

FCBDIR

Specify FCBDIR when outputting a control record that has an FCB name specified in a program and using FCB. For information about FCB, refer to "7.1.6 FCB".

Example

When specifying FCB1 after /home/usr1/fcbe:

```
$  FCBDIR=/home/usr1 ; export FCBDIR
```

FOVLDIR

Specify the absolute path name of the form overlay pattern file storage destination folder. For information about FOVLDIR, refer to "7.1.7 Form Overlay Patterns".

Example

When specifying OVL1 after /home/usr1/kol5:

```
$ FOVLDIR=/home/usr1 ; export FOVLDIR
```

File-identifiers Specified with ASSIGN Clause

If specifying a file-identifier for the ASSIGN clause in the control entry, specify the file-identifier with one the following full path names as the value for the environment variable:

- For print files without FORMAT clause: Output destination file

- For print files with FORMAT clause: Printer information files

Example

When allocating prtfile after /home/usr1 for output destination by a print file without FORMAT clause that the description in the program is "ASSIGN TO PRTFILE":

```
$  PRTFILE=/home/usr1/prtfile ; export PRTFILE
```

PRINTER-n (n=1..9)

If specifying PRINTER-n for the ASSIGN clause in the file control entry by a print file without FORMAT clause, set PRINTER-n as the environment variable with the path name of the output destination as its value. However, PRINTER-n cannot be used with Bash.

The following shows an execution example with C shell.

Example

When allocating prtfile after /home/usr1 for output destination by a print file without FORMAT clause that the description in the program is "ASSIGN TO PRINTER-1":

```
%  setenv PRINTER-1 /home/usr1/prtfile
```

LD_LIBRARY_PATH

Specify storage directories for libraries offered by systems that need for print processing.

Note

When adding the storage directory /opt/FPFORM/lib for PowerFORM RTS in LD_LIBRARY_PATH:

```
$  LD_LIBRARY_PATH=/opt/FPFORM/lib:$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

7.1.5 Print Information Files

Print files are text files used when outputting forms using a print file without a FORMAT clause.

Set some status control information about forms output in the print information file.

In the following cases, print information files must be specified in the print file without a FORMAT clause:

- Changing page formats.

Specifying a Print Information File

To specify print information files, specify the print information file's path name as the value for the CBR_PRT_INF environment variable. In this case, the same print information file corresponds to all files in the run unit regardless of the ASSIGN clause description.

Print information files can be specified for each print file if describing a file-identifier, a file-identifier literal, a data- name, or PRINTER-n in the ASSIGN clause.

If specifying a print information file for each print file, specify ",,INF (the path name for print information file)" following a path name for a print file for output. Note that if specifying a print information file and a font table at the same time, separate the print information file and font table path names with commas (,).

```
"the-print-file-path-name,,INF(the-print-information-file-path-name),FONT(the-font-table-path-name)"
```

If specifying a relative path in the path name for print information files, search the relative path from a current directory.

If the specification for each print file and for an entire run unit exists, the specification for each print file is prioritized.

Example

Example 1

When allocating prtfile after /home/usr1 for output destination and insatsu.inf as a print information file with the print file when the description in the program is "ASSIGN TO PRTFILE":

```
$  PRTFILE="/home/usr1/prtfile,,INF(/home/usr1/insatsu.inf)" ; export PRTFILE
```

Example 2

When allocating insatsu.inf after /home/usr1 as a print information file and fonttab as a font table with the print file when the description in the program is "ASSIGN TO PRTFILE":

```
$  PRTFILE=",,INF(/home/usr1/insatsu.inf),FONT(/home/usr1/fonttab)" ; export PRTFILE
```

Note

- "INF" and "FONT" must be described in capital letters.
- If the output destination specification omitted, the prtout control statement in the print information file must be specified.

Description Formats of Print Information Files

Execute analysis of a print information file when an OPEN statement for a print file is executed. The contents are valid until a CLOSE statement for the print file is executed.

Print information files are text files configured with the following control statements:

- printer control statement.
- papersize control statement.
- prtform control statement.
- fcbname control statement.
- prtout control statement.

The following is an example of the description for the print information file and description formats for each control statement:

```
printer      PS2
papersize    ltr
prtform      l
fcbname      fcb1
prtout       /home/usr1/prtfile
```

Note

- Spaces and tab characters are used for delimiting keywords and parameters in each control statement.
- Lines preceded by a hash mark (#) are assumed as comment lines.
- If a control statement is specified more than once, the latest specification is valid.

printer Control Statement

The printer control statement specifies the kind of datastream.

Keyword	Parameter
printer	kind of datastream

Parameters

Specify the kind of datastream with one the following character strings:

Value	Meaning
PS1	PostScript level 1
PS2	PostScript level 2

Defaults

If the printer statement is not specified, the default value is PS1 (PostScript level 1).

papersize Control Statement

The papersize control statement specifies a paper size for the default.

Keyword	Parameter
papersize	Paper size for the default

Parameters

Specify the paper size for the default with one the following character strings:

Value	Meaning
a3	A3 size
a4	A4 size
a5	A5 size
b4	B4 size
b5	B5 size
ltr	LETTER size

Defaults

If the papersize control statement is not specified, the default value is ltr (LETTER size).

Note

Continuous-form paper cannot be specified.

prtform Control Statement

The prtform control statement specifies a print format for the default.

Keyword	Parameter
prtform	Print format for the default

Parameters

Specify the print format for the default with one the following character strings:

Value	Meaning
p	Portrait

Value	Meaning
l	Landscape
pz	Portrait compression
lz	Landscape compression
lp	LP compress print

Defaults

If the data stream is not PostScript level 1 and no print format is specified in the print information file and I control record, portrait ("p") is assumed. If the data stream is PostScript level 1 and no print format is specified in the print information file and I control record, LP compressed print ("lp") is assumed.

Note

Actual print results depend on the printer device functions.

fcbname Control Statement

The fcbname control statement specifies FCB name to be used.

Keyword	Parameter
fcbname	FCB name for the default

Parameters

Specify the FCB name to be used with up to 4 alphanumeric characters.

Defaults

If the fcbname control statement is not specified, it is assumed that FCB is not specified and the default value for FCB is adopted. For information about the default value for FCB, refer to "7.1.6 FCB".

Note

When specifying the fcbname control statement, the environment variable FCBDIR must be specified.

prtout Control Statement

Specify output destination for print data. The specification of the prtout control statement gives priority to an ASSIGN clause description in programs. Specify the prtout control statement when you want to change an output destination specified in the ASSIGN clause or the specification of the output destination in the ASSIGN clause has been omitted.

Keyword	Parameter
prtout	Output destination for print data

Parameters

Specify the path name of the print files to be output.

Defaults

If the prtout control statement is not specified, the specification for the ASSIGN clause description is valid. If the prtout control statement is omitted and there is also no valid specification in the ASSIGN clause, an error occurs.

Note

Direct print (ASSIGN TO PRINTER) to a printer cannot be specified with the prtout statement. Also, print information file names (INF(...) and font table names (FONT(...) cannot be specified in this statement.

7.1.6 FCB

Use FCB to change the number of lines on one page, line spaces, and the column start position.

Specifying FCB

To use FCB, first create FCB under the directory called fcbe (which is a fixed directory).

Specify FCB name in the I control record and set a directory path name of an upper fcbe directory when a program is executed in the FCBDIR environment variable. Also the FCB name used by the default can be specified using the methods below.

The default FCB is valid when FCB is not specified with the I control record.

- Print files without FORMAT clause:
  - The fcbname control statement of print information files.
- Print files with FORMAT clause:
  - Environment variable CBR_FCB_NAME.

The values in the following table shows the print information if FCB is omitted:

Table 7.5 the default values for FCB

Print information	Print file without FORMAT clause	Print file with FORMAT clause	
		Without form descriptor	With form descriptor
Form size.	11-inch	11-inch	The value specified in the form descriptor.
line space.	6LPI	6LPI	
Number of lines.	66	66	
Column start position.	4	4	*1

*1: The value is either 4 for output of a floating partition, or the value specified in a form descriptor for output of a fixed partition.

Note

Specifying FCB does not determine the sizes or the orientation of forms offered by a printer.

Specification of the form sizes and the print formats is determined using the I control record. For more information, refer to "7.1.14 I and S Control Records".

FCB Formats

FCB are text files composed of the following two types of control statements:

- lpi control statements.
- print control statements.

Each FCB statement format is explained as follows:

- Spaces and tab characters are used for delimiting keywords and parameters in each control statement.
- Lines preceded by a hash mark (#) are assumed as comment lines.

lpi Control Statement

Keyword	Parameter
lpi	Line spaces, line numbers

Parameters

Line spaces

Specify line spaces with a unit of line spaces (1/7200 inches).

Only the following values can be specified in the lpi control statement:

Line space	Value
6LPI	1200
8LPI	900
12LPI	600

Line numbers

Specify the line numbers that the line spaces can be used on. If the value for line spaces * line numbers exceeds the form size specified in the print control statement, an FCB error occurs.

print Control Statement

Keyword	Parameter
print	Column start position [a form size]

Parameters

Column start positions

Specify the column start position in pages.

Form sizes

Specify portrait length for forms (in units: 1/7200 inches). If omitted, the default value is 79200 (11 inches). Form length is rounded up by the 3600/7200 units (0.5 inches).

The table below shows the values indicated physical forms for cut-sheets length by in units of 1/7200 inches. Note that these values contain unprintable areas. For this reason, if specifying actual values, unprintable areas must be considered. If the actual form length is shorter than the form length specified by FCB, data before the data may be lost.

Form size	Form direction	
	Portrait	Landscape
A3	119000	84200
A4	84200	59500
A5	59500	42100
B4	103200	72900
B5	72900	51600
LETTER	79200	61200

Note

The position of CHANNEL specified by FCB cannot be set in this system. A WRITE statement specified mnemonic-names corresponded to CHANNEL-02 to CHANNEL-12 operates the same as a WRITE statement with a LINE phrase in the ADVANCING 1. A WRITE statement specified mnemonic-names corresponded to CHANNEL-01 processes page alignment.

An example of FCB Description

The following is an example of an FCB description with the following conditions:

```
Form size: A4
Form direction: Landscape
Column start position: Line 1
Line spacing: 6lpi
```

```
print    1  55900
lpi   1200     46
```

Explanation:

Physical length for form is 59500/7200 inches. However, if unprintable area that upper and lower sides of a form are summed is 3600/7200 inches (=1/2 inch), the unprintable area can be calculated as follows:

```
59500/7200 - 3600/7200 = 55900/7200
```

Calculate printable line numbers per page with line spaces: 6lpi (1200/7200 inches) for printable area. The value is:

```
(55900/7200) / (1200/7200) = 46.58
```

The size for printable area depends on the printer.

7.1.7 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 easily by overlaying a page of output data with a FORM overlay pattern.

A FORM overlay pattern can be generated easily from a screen image by the 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.

FORM overlay patterns (KOL5 or KOL6 format) can be printed using a print file with a FORMAT clause.

To use a FORM overlay pattern, store the FORM overlay pattern in the directory in which is under kol5 (which is a fixed directory). For the print files with a FORMAT clause, specify the storage directory of the FORM overlay pattern in the printer information files.

If executing a form print using a FORM overlay pattern by the I control record, the FORM overlay pattern name can be a maximum of 4 alphanumeric characters with no extension. Specify the same name as the FORM overlay pattern file name in the I control record.

For more information about how to create overlay patterns, refer to "PowerFORM Runtime Reference" and "PowerFORM on-line help". For printing forms when using FORM overlay patterns, see "7.3 How to Use a Control Record".

7.1.8 Form Descriptors

When forms are designed with 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.

PowerFORM Runtime is required when printing forms using form descriptors. PowerFORM Runtime requires a printer information file used by PowerFORM Runtime. For details of printer information files, refer to the "PowerFORM Runtime Reference". For information on how to generate form descriptors, refer to the "PowerFORM Runtime Reference" or "PowerFORM on-line help". For information about printing forms by using form descriptors, see "7.4 Using Print Files with a FORM Descriptor".

For notes on printing form descriptors created by PowerFORM , refer to the "PowerFORM Runtime Reference".

Note

If creating form descriptors, names set/referred with COBOL program you need to note the following:

- File names, except for the extension of file descriptor used in COBOL, must be an alphanumeric character string consisting of up to 8 characters and beginning with an alphabetic character.
- Specify item group names and partition names using up to 6 single-byte alphanumeric characters.
- Specify item names according to the rules of user-defined words used in COBOL.

7.1.9 Font Tables

The font table is a text 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. Optional fonts can be associated for font numbers (FONT-nnn) specified in a PRINTING MODE clause of particular name for paragraphs using font tables.

Specifying Font Tables

To specify font tables, specify the font table file path name as the value for the CBR_PRINTFONTTABLE environment variable. In this case, the same font is corresponded to all print files in run unit regardless of an ASSIGN clause description.

Font tables can be specified for each print file if describing a file-identifier, a file-identifier literal, a data- name, or PRINTER-n in the ASSIGN clause.

If specifying a font table for each print file, specify ",,FONT (the font table path name)" following a path name for a print file for output. Note that if specifying a print information file and a font table at the same time, separate them with commas (,).

```
"the-print-file-path-name,,INF(the-print-information-file-path-name),FONT(the-font-table-path-name)"
```

If specifying a relative path in the path name for font tables, search the relative path from a current directory.

If the specification for each print file and for an entire run unit exists, the specification for each print file is prioritized.

Example

Example 1

When allocating prtfile after /home/usr1 for output destination and fonttab as a font table with the print file that description in the program is "ASSIGN TO PRTFILE".

```
$  PRTFILE="/home/usr1/prtfile,,FONT(/home/usr1/fonttab)" ; export PRTFILE
```

Example 2

When allocating insatsu.inf after /home/usr1 as a print information file and fonttab as a font table with the print file that description in the program is "ASSIGN TO PRTFILE".

```
$  PRTFILE=",,INF(/home/usr1/insatsu.inf),FONT(/home/usr1/fonttab)" ; export PRTFILE
```

Note

- "INF" and "FONT" must be described in capital letters.
- If the output destination specification omitted in a print file without a FORMAT clause, the prtout control statement in the print information file must be specified.
- For print files with a FORMAT clause, you must specify the printer information file name.

Font Table Format

The following shows the font table format:

```
[Font number]             Section name (it must be specified for each font number).
FontName=font-face-name   Specify a typeface name of the font (optional).
Style={ R | B | I | BI }  Specify a print style (optional).
```

Note

- Lines preceded by a hash mark (#) are assumed as comment lines.
- If the same font number is specified more than once, the latest specification is valid.

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 font set by the system.

Specify the font face name using up to 32 bytes of alphanumeric characters or national characters.

Specify font names that print devices can be identify directly as the font face name. Check the printable font in the printer manuals to specify it correctly.

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

Note

Only when the level 2 for PostScript is valid in the print files with FORMAT clause.

Notes on Using Font Tables

Specifying fonts by the font number is valid only when data stream is the level 1 for PostScript.

Note the following if the data stream is PostScript:

- Styles specified in Style=, may not be valid. Because the fonts are provided for each normal style in the PostScript printer, specify font face names that have style attributions to be used in FontName=.

Example

Example 1

When using the bold Courier (Courier-Bold):

```
FontName=Courier-Bold
```

Example 2

When using the italic Courier (Courier-Oblique):

```
FontName=Courier-Oblique
```

- The font table template is stored in the directory below. If using font tables, customize this template to create a font table. If allocating fonts used in the system, the processes may not be operated correctly. So that if allocating new fonts, use unused numbers (301 to 800 are recommended).

  [Storage directory of the template]

  COBOL installation directory/config/template/C/fonttable

- The font numbers and font names below are used as the same values. If you change these font numbers, an abnormality for the print result occurs.
  - FONT-001 and MINCHOU
  - FONT-002 and MINCHOU-HANKAKU
  - FONT-003 and GOTHIC
  - FONT-004 and GOTHIC-HANKAKU

Example

An example for creating a font table:

```
[FONT-301]
FontName=Courier-Bold
[FONT-302]
FontName=Courier-Oblique
```

7.1.10 Special Registers

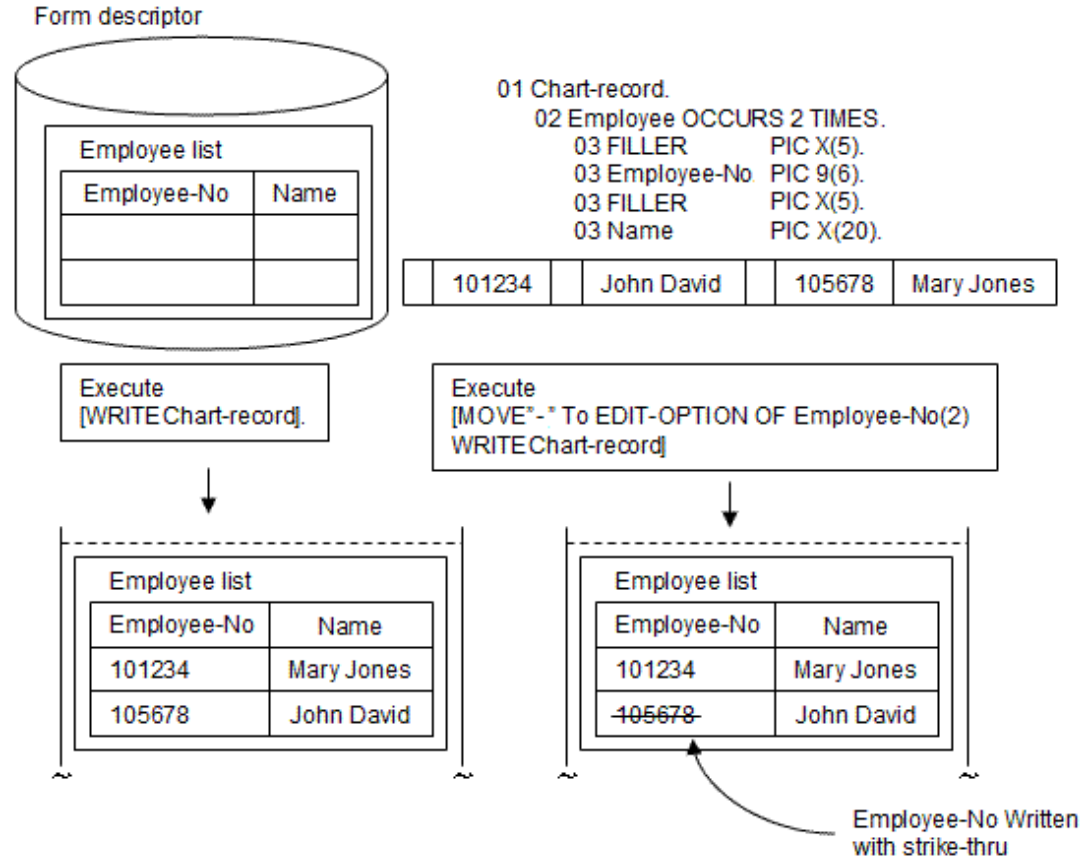
You can modify output data attributes for print files with the FORMAT clause, by using COBOL special registers.

The special registers for form functions are:

- EDIT-MODE: Specifies whether to use output processing or not.
- EDIT-OPTION: Specifies the underline/killer bars 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 "PowerFORM Runtime Reference".

Figure 7.1 Special Registers



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.

7.1.11 How to Determine Print File

COBOL offers two ways to create print files: use a print file without a FORMAT clause or use a print file with a FORMAT clause.

The compiler recognizes the different styles of print files by recognizing the presence of specific elements in the source program.

Specific Elements

[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.

File Type Combinations Determined by the Specific Elements

The table below shows the file type determined by the presence of the above elements [1] through [5]:

File type	[1]	[2]	[3]	[4]	[5]
Print file without FORMAT clause	C	A (*)	A (*)	A (*)	A (*)

File type	[1]	[2]	[3]	[4]	[5]
Print file with FORMAT clause	A	C	C	C	B

* : The presence of [2], [3], [4], or [5] determines that the file is a print file without a FORMAT clause.

A : Element that determines the file type.

B : Element can be included but does not affect the file type.

C : Cannot be included in that file type.

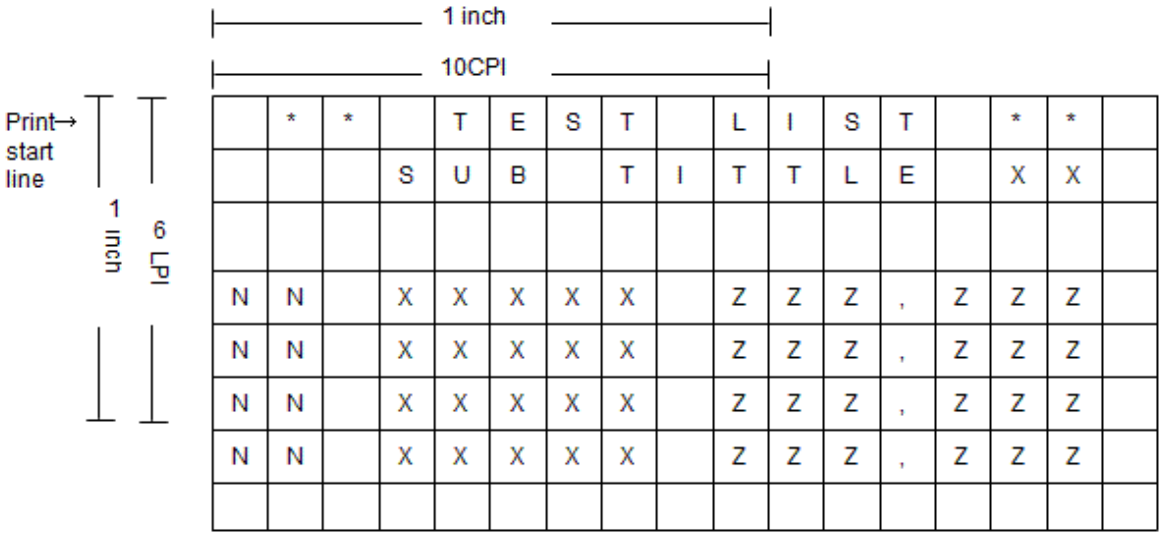
7.1.12 Note Forms Design

The concepts of lines and columns are indispensable for printing COBOL forms. 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 print 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



Description of FCB file:

```
lpi          1200    66
print        1
```

COBOL program:

```
    SPECIAL-NAMES.
      PRINTING MODE PM1 IS FOR ALL
        AT PITCH 10.00 CPI.
*>              :
  DATA DIVISION.
*>              :
     01 PRINT-RECORD PIC X(80)
```

```
                    CHARACTER TYPE IS PM1.
*>              :
  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 below, 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 file.

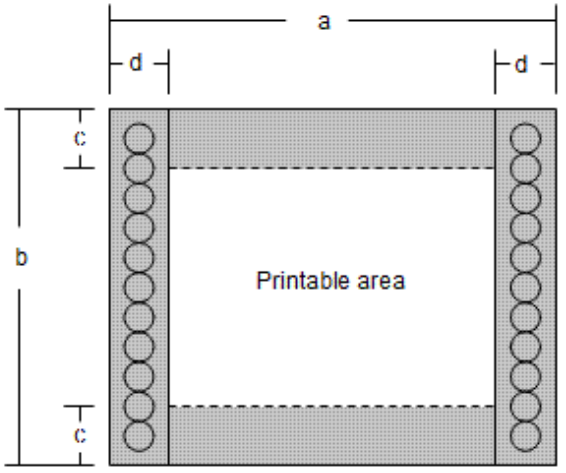
The FCB file specifies the column start position, form length, line spacing (LPI), and the number of lines that the line spacing (LPI) is valid. The number of lines that can be printed on a form is determined by the form length and line spacing defined by the FCB file.

An example of the FCB file 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 it in unit of 1/7200 inches)

Example of FCB file definition

```
lpi          1200    66
print        1       79200
```



The settings made before shipment are as follows (adjustable):

a: Form size (horizontal)
b: Form size (vertical)
c: Unprintable area
d: Unprintable area

Some printers, due to their hardware specification, have physically unprintable areas on the upper, lower, right, and left sides of each form. The lp system filters 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.

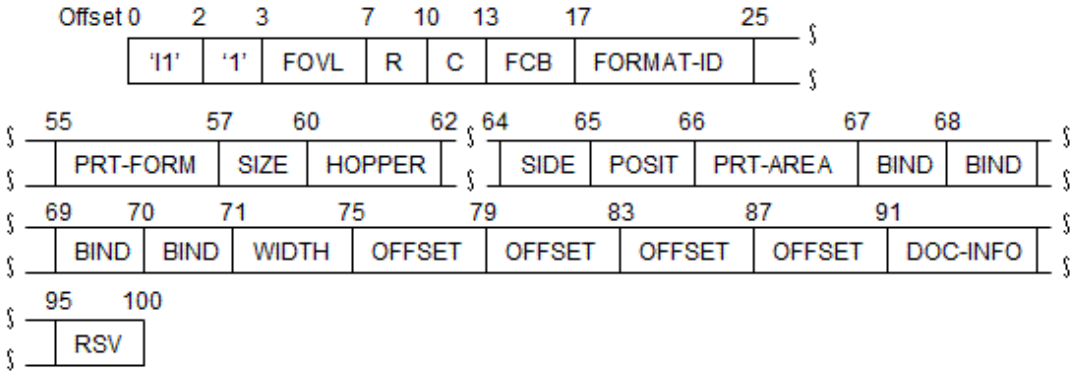
7.1.14 I and S Control Records

There are I control records and S control records in control records. Formats of the control records are described below.

The valid functions depend on the printer device. Refer to the printer device's instruction manual.

I Control Records

The following shows a format of I control records.




```
01  I-control-record.
      02  Identifier                PIC  X(2)  VALUE  "I1".
      02  Form                      PIC  X(1)  VALUE  "1".
      02  Overlay-name              PIC  X(4).                 *>FOVL
      02  Print-count               PIC  9(3).                 *>R
      02  Copy-count                PIC  9(3).                 *>C
      02  FCB-name                  PIC  X(4).                 *>FCB
      02  Form-descriptor           PIC  X(8).                 *>FORMAT-ID
      02                            PIC  X(30).
      02  Print-form                PIC  X(2).                 *>PRT-FORM
      02  Form-size                 PIC  X(3).                 *>SIZE
      02  Form-feed-port            PIC  X(2).                 *>HOPPER
      02                            PIC  X(2).
      02  Print-face-specification  PIC  X(1).                 *>SIDE
      02                            PIC  X(1).
      02  Print-inhibit-area        PIC  X(1).                 *>PRT-AREA
      02  Binding-margin-direction.
            03                      PIC  X     OCCURS 4 TIMES.  *>BIND
      02  Binding-margin-width      PIC  9(4).                 *>WIDTH
      02  Print-origin.
            03                      PIC  9(4)  OCCURS 4 TIMES.  *>OFFSET
      02                            PIC  X(9)  VALUE  SPACE.    *>RSV
```

- FOVL

  Specifies a FORM overlay pattern name to be used. Note that only FORM overlay patterns can be specified. If an overlay group is specified, an error occurs.

  NOTE:

  A FORM overlay pattern can be used only when using a print file with a FORMAT clause.

- R

  Specifies the number of print times of FORM overlay pattern (0 to 255).

- C

  Specifies the number of copies of each page (0 to 255).

Note

The copy count setup is invalid for dual-side printing. If two or more are specified for the copy count, the operation is not guaranteed. Specify 0 or 1 for the copy count with dual-side printing.

- FCB

  Specifies the FCB name to be used.

- FORMAT-ID

  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.

- PRT-FORM

  Specifies a print form. The following values are available:

  "P" (Portrait mode), "L" (Landscape mode), "LP" (Line printer mode), "PZ" (Compressed portrait mode), or "LZ" (Compressed landscape mode)

  Note that compressed printing may be unable to be processed, depending on the printer.

Note

The print form (form orientation) for forms provided by printers is determined by this field. If this field is omitted, the default values are as follows:

- For print files without FORMAT clause:

  The value set for the print information files -> the default value set by COBOL or the lp system.

- For print files with FORMAT clause:

  The value set for form descriptors -> the value set for the printer information files -> the value set for the printer.

Creating and specifying the FCB file depends on if the print form used actually needs it. For more information, refer to "7.1.6 FCB".

- SIZE

  Specifies a form size. The following sizes are available:

  "A3", "A4", "A5", "B4", "B5", or "LTR" (letter)

  Note that a form size may be unavailable, depending on the printer.

Note

The form size (form type) for forms provided by printers is determined by this field. If this field is omitted, the default values are as follows:

- For print files without FORMAT clause:

  The value set for the print information files -> the default value set by COBOL or the lp system.

- For print files with FORMAT clause:

  The value set for form descriptors -> the value set for the printer information files -> the value set for the printer.

Creating and specifying the FCB file depends on if the form size used actually needs it. For more information, refer to "7.1.6 FCB".

- HOPPER

  Specify a hopper. The following values are available:

  "P1" (Main hopper 1), "P2" (Main hopper 2), "P3" (Main hopper 3), "P4 (Main hopper 4)", "S" (Subhopper), "P" (Any hopper)

  Note that a hopper may be unavailable, depending on the printer.

- SIDE

  Specifies a print side. The following values are available:

  "F" (Single-side printing) or "B" (Dual-side printing)

Note

If the dual-side printing is specified, the copy count setup is invalid.

- PRT-AREA

  Specify to inhibit printing (L) or not (N).

- BIND

  Specifies a binding margin direction when multiple continuous output pages are bound. The following values are available:

  "L" (Left), "R" (Right), "U" (Up), or "D" (Down)

Note

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

  Specifies a binding margin width in the range of 0 to 9999 (in units of 1/1440 inch).

- OFFSET

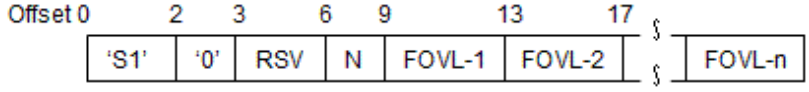
  Specifies the print origin in the range of 0 to 9999 (in units of 1/1440 inch).

- RSV

  This is the system reserved area. Keep the RSV blank.

S Control Records

The following shows a format of S control records.




```
01  S-control-record.
      02  Identifier              PIC  X(2)  VALUE  "S1".
      02  Form                    PIC  X     VALUE  "0".
      02                          PIC  X(3)  VALUE  SPACE.    *>RSV
      02  Overlay-sequence-count
                                  PIC  9(3).                 *>N
      02  Overlay-name-1          PIC  X(4).                 *>FOVL-1
*>                :
      02  Overlay-name-n          PIC  X(4).                 *>FOVL-n
```

- RSV

  This is the system reserved area. Keep the RSV blank.

- N

  Specifies the number of FORM overlay pattern names for FOVL-n.

- FOVL-n

  A FORM overlay pattern is repeated for the number specified with the I control record using this order. However, the system only uses the first FORM overlay pattern name.

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 times when the record is output and when the next S control record or I control record is output.

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.

Table 7.6 Whether the I/S control functions can be specified or not

		Character strings identified by printer devices
I/S control functions	FORM overlay module name	A (*1)
	Number of printings	A (*2)
	Copying number	A
	FCB name (FCB)	A
	Form descriptors name	B
	Copy modification module name	B
	Copy modification starting number	B
	Copy modification character arrangement table number	B
	Form identifier name	B
	Character arrangement table and additional character set	B
	Dynamic load	B
	Offset stack	B
	Print format	A
	Form size	A (*4)
	Form supply port	A (*3)(*4)
	Form output port	B
	Printing face specification	B
	Printing face positioning	B
	Unprintable area	B
	Binding margin direction	A
	Binding margin with and print starting origin	A

A: can be specified

B: cannot be specified

*1 : This cannot be specified when using a print file without a FORMAT clause.

*2 : The specification of number of printings is assumed with the same value specified for the copying number.

*3 : Only main hopper 1 and main hopper 2 can be specified.

*4 : This may be invalid depending on the printer device. In this case, specify options by the lp command. However, note that the options that can be specified depend on the specifications of filters and devices.

7.2 How to Print Data in Line Mode Using a Print File

This section explains how to print data in line mode using a print file without a FORMAT clause. A sample program using a print file is provided to refer to.

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 for using a Print File without FORMAT

This section details program descriptions using a print file without FORMAT clause for each COBOL division.

```
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}].
     data 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.
```

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, write the relationship 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

The following table lists information required to specify a file control entry:

Table 7.7 Information to be specified in a file control entry

	Location	Information Type	Details and Use of Specification
Required	SELECT clause	File name	Specify 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 PRINTER or the same medium information as the record sequential file. When PRINTER is specified, data is output to the standard writer of the system. (*1)
Optional	ORGANIZATION clause	Keyword indicating file organization	Specify SEQUENTIAL.
	FILE STATUS clause	Data-name	Specify the data-name defined as a double-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. The input-output execution result is set for this data-name. (*2)

*1: For information on specifying other methods, see "6.2.1 Defining Record Sequential Files".

*2: For information on what values to 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. The following table lists information required to write a file description entry:

Table 7.8 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.
	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 an input-output statement in the following sequence:

1. OPEN statement with OUTPUT phrase: Starts printing.
2. WRITE statement: Outputs data.
3. CLOSE statement: Stops printing.

OPEN and CLOSE Statements

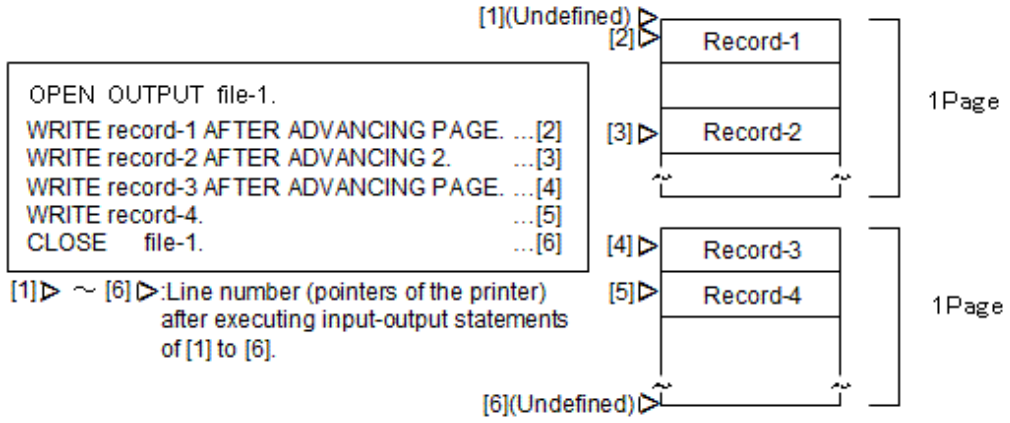
Use an OPEN statement once at the start of printing and a CLOSE statement once at the end of printing.

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:



Note

- When writing "WRITE A FROM B" in the WRITE statement specified FROM, 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.
- Executing the WRITE statement with AFTER ADVANCING PAGE specified immediately after executing the OPEN statement does not align pages.

Input-Output Error Processing

For information about how to detect I-O errors and execution results when I-O errors occur, 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

The operation to execute a program using a print file is depends on the description contents of an ASSIGN clause in the file control entry. The following table shows the necessary operation before and after executing programs for each description content of an ASSIGN clause:

Table 7.9 Necessary operation for the program execution

Operation for the Program Execution		Description of ASSIGN Clause		
		PRINTER	File-identifier PRINTER-n	File-identifier Literal or Data Name
Before execution	Setting file-identifier as the environment variable with the path name of the output destination as its value.	-	Required	-
	Setting the CBR_LP_OPTION environment variable.	Optional	-	-

Operation for the Program Execution		Description of ASSIGN Clause		
		PRINTER	File-identifier PRINTER-n	File-identifier Literal or Data Name
	Setting the CBR_PRT_INF environment variable.	Optional	Optional	Optional
After execution	Starting the lp command.	- *1	Required	Required
Output destination		Standard writer of the system	File	File

*1 If PRINTER is specified in the ASSIGN clause, the lp command is automatically started when the CLOSE statement is executed. For details on options specified in the lp command, see "CBR_LP_OPTION" in "7.1.4 Setting Environment Variables".

For information about each environment variable setting, refer to "7.1.4 Setting Environment Variables".

Information

For information about each environment variable setting, refer to "7.1.4 Setting Environment Variables".

7.3 How to Use a Control Record

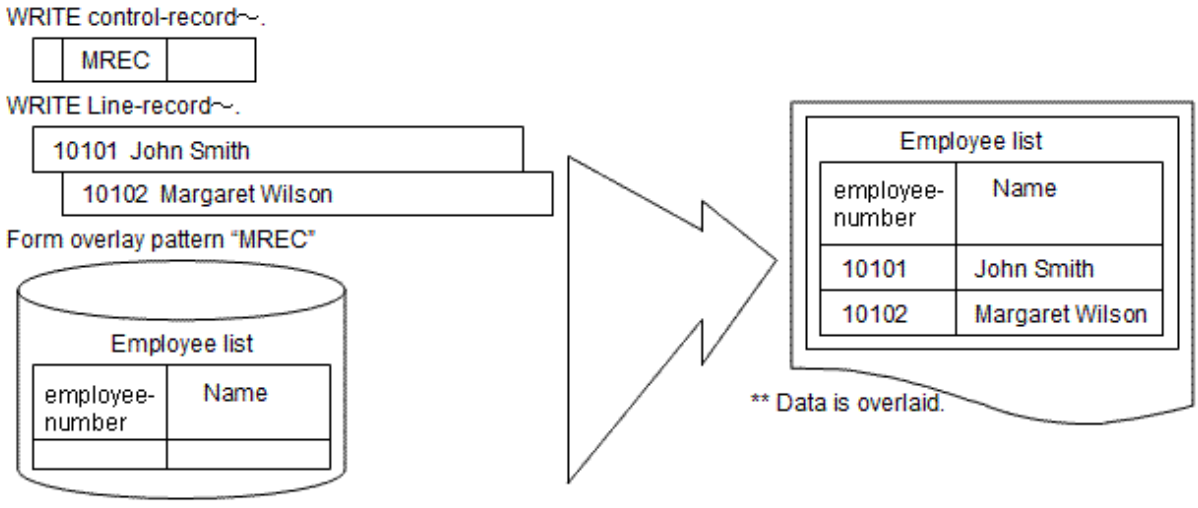
This section explains how to execute overlaid printing using a FORM overlay pattern.

7.3.1 Outline

By using a FORM overlay pattern with a print file, forms can be printed easily. Use a control record to employ a FORM overlay pattern.

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 and printed 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 "7.1.3 Print Characters" and also refer to the "PowerFORM Runtime Reference" or "PowerFORM Online help".



Note

This product does not offer the overlaid printing using a print file without a FORMAT clause.

7.3.2 Program Specifications for using FORM Overlay Patterns

This section describes the code required in each of the COBOL divisions when you use FORM overlay patterns.

```
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}].
       record description entry
 01  control-record-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-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] [AFTER ADVANCING  ...].]
    CLOSE  file-name.
 END PROGRAM  program-name.
```

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, write the relationship 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 outputting 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 7.7 Information to be specified in 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 7-8: 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 a line record size smaller than 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 "7.1.3 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 in "7.1.14 I and S Control Records".

PROCEDURE DIVISION

Execute the input-output statements in the following sequence:

1. OPEN statement with OUTPUT phrase: 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 for using a Print File without FORMAT".

To write a control record, write the mnemonic-name related to the function-name CTL in the ADVANCING phrase.

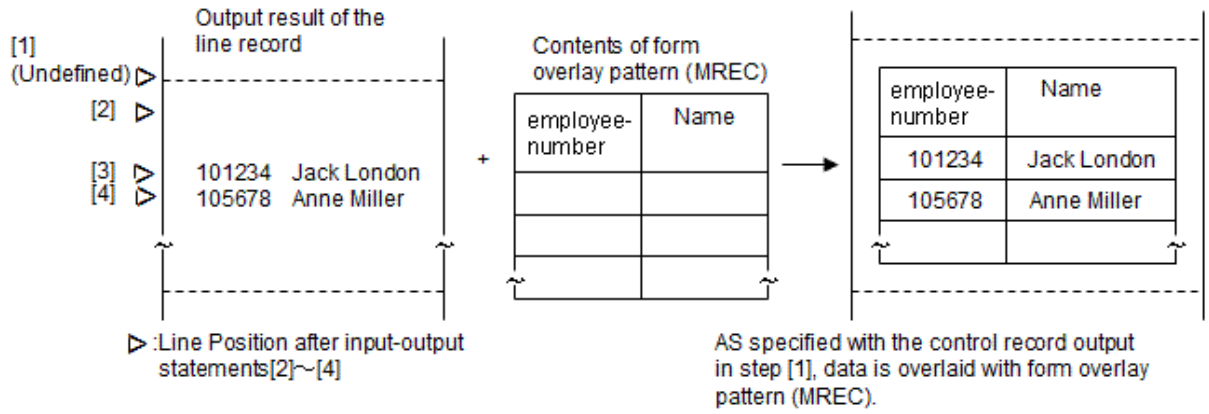
To overlay data created by a line record with a FORM overlay pattern, write a control record with a FORM overlay pattern name specified.

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  control-record.
*>   :
      02  FOVL       PIC X(4).
*>   :
     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]
*>       :
```



Note

When writing "WRITE A FROM B" in the WRITE statement specified FROM, 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.

Input-Output Error Processing

For information about how to detect I-O errors and execution results when I-O errors occur, 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

The operation to execute a program using a print file depends on the description contents of an ASSIGN clause in the file control entry. The following table shows the necessary operation before and after executing programs for each description content of an ASSIGN clause:

Table 7.10 Necessary operation for the program execution

Operation for the Program Execution		Description of ASSIGN Clause		
		PRINTER	File-identifier PRINTER-n	File-identifier Literal or Data Name
Before execution	Setting file-identifier as the environment variable with the path name of the output destination as its value.	-	Required	-
	Setting the CBR_LP_OPTION environment variable.	Optional	-	-
	Setting the CBR_PRT_INF environment variable.	Optional	Optional	Optional
	Setting the CBR_FCB_NAME environment variable.	Optional	Optional	Optional
	Setting the FCBDIR environment variable.	Optional	Optional	Optional
	Setting the FOVLDIR environment variable.	Optional	Optional	Optional
After execution	Starting the lp command.	-	Required	Required
Output destination		Standard writer of the system	File	File

Information

For information about each environment variable setting, refer to "7.1.4 Setting Environment Variables".

Note

- The path name of the FORM overlay pattern storage directory must be set in the FOVLDIR environment variable during program execution. To use an FCB, the path name of the FCB storage directory must be set in advance in the FCBDIR environment variable.
- Using the lp command to output the print file written to a disk file requires no special option.
- For details on the supported range of FORM overlay pattern output, refer to "PowerFORM Runtime Reference".
- If the data stream is PostScript level 2, the overlay characters defined in a FORM overlay pattern in the KOL6 format are printed in the following fonts:
  - National characters: Mincho
  - Alphanumeric characters: Gothic

7.4 Using Print Files with a FORM Descriptor

This section explains how to use partitioned form descriptors by using a print file with a FORMAT clause.

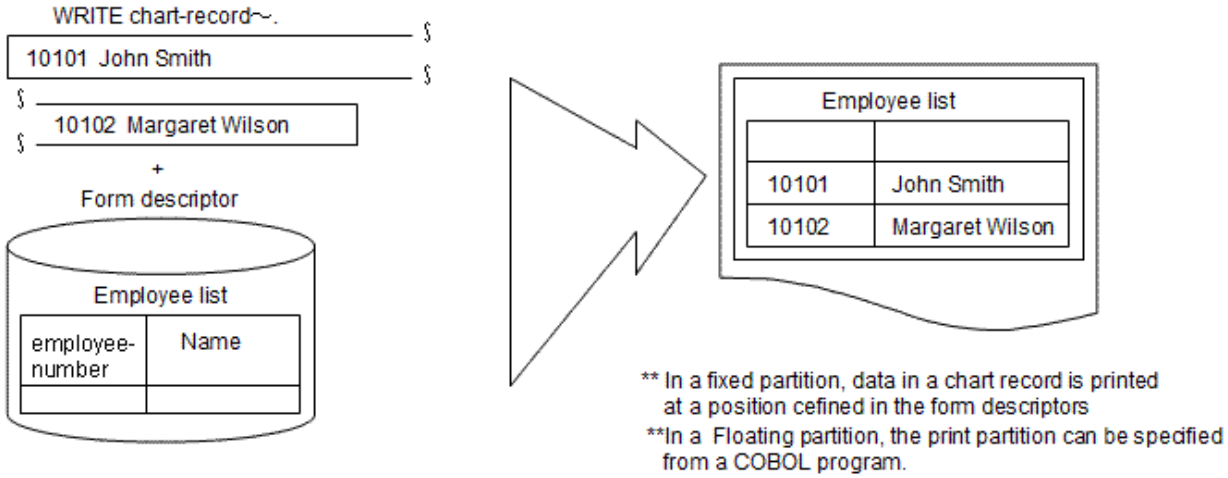
For a sample program using a print file with a FORMAT clause is provided to refer to.

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. 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 "7.1.3 Print Characters", the "PowerFORM Runtime Reference", and "PowerFORM Online help".



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.
- The forms that contain the encoding of the following national item cannot be output.
    - Big endian
    - UTF-32

7.4.1.1 Form 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.

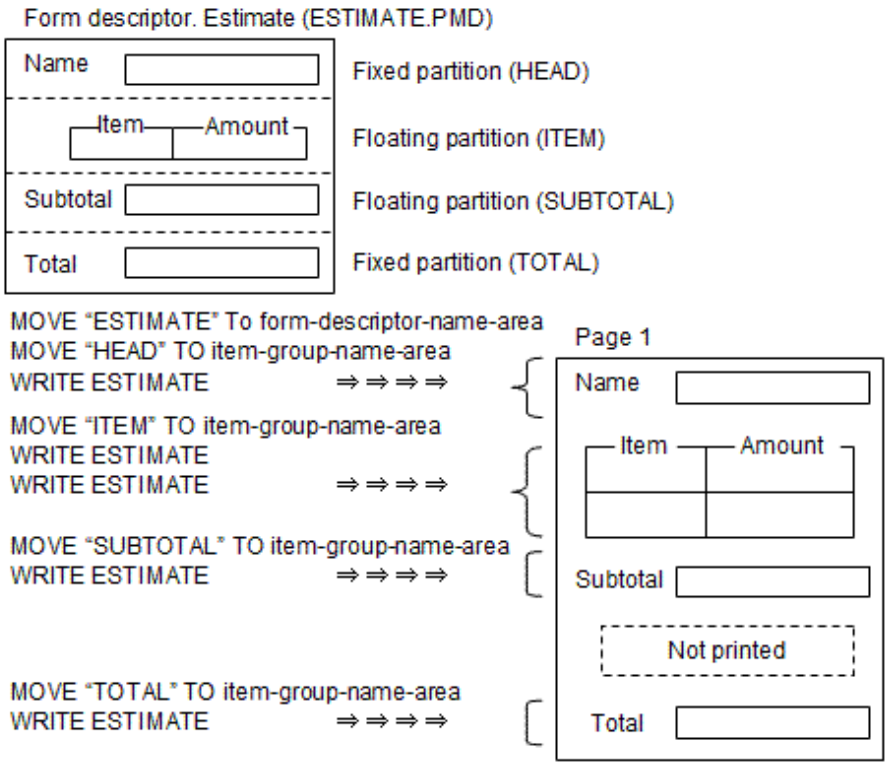
Floating Partitions

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

Output Example

The following shows the WRITE statement for chart records and gives an output example of fixed and floating partitions to be output:



The floating partition is printed out according to 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

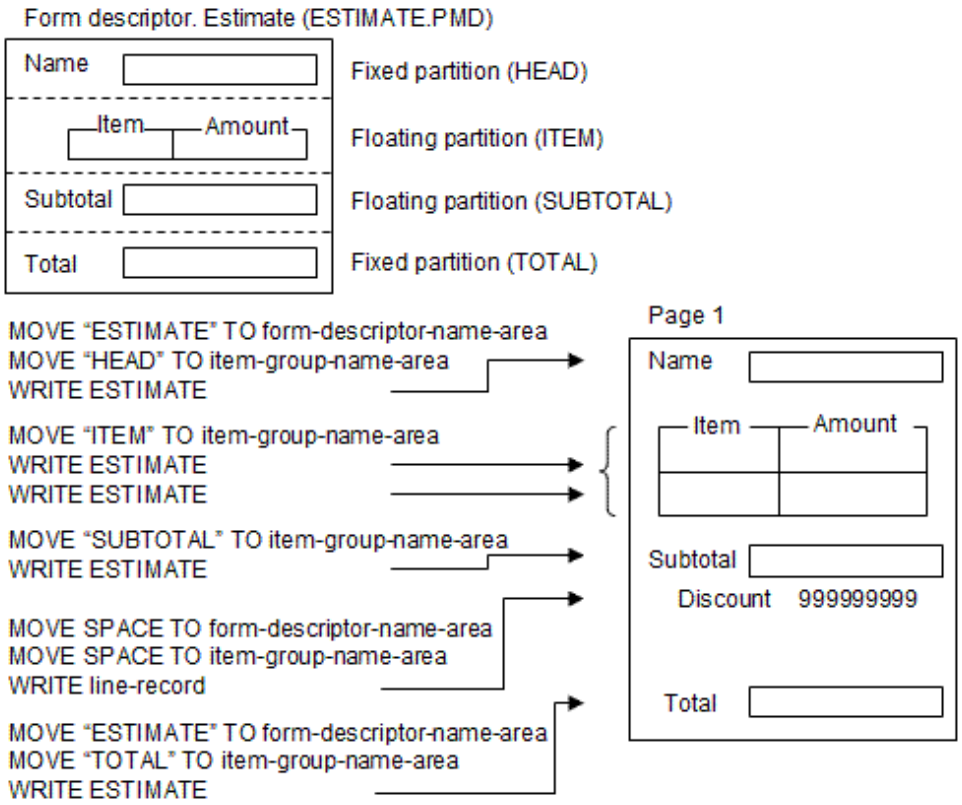
The partitions defined in the form descriptor can be printed out using the chart records. Also, normal line records can be output to any position of the same page.

The partitions and line records are printed out according to the WRITE statement execution sequence (or after the number of lines specified by the ADVANCING phrase if any has been fed).

Example

Output 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.



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.

```
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}].
       record-description-entry
```

```
 01  control-record-name.
       record-description-entry
     COPY  form-descriptor-name  OF  XMDLIB.
(01  chart-record-name.    ) (*)
(      record-description-entry   )
 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      [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.
    MOVE   SPACE  TO  item-group-name-area.
    WRITE  line-record-name  [FROM data-name] [AFTER ADVANCING  ...].
    CLOSE  file-name.
    EXIT PROGRAM.
END PROGRAM  program-name.
```

* Note that the phrases in parentheses above indicate the COPY statement expansion.

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

Define a print file in a file control entry. The following table lists the information required to write a file control entry.

Table 7.11 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-identifier to assign a printer information file to be used by PowerFORM 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.
	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	Specify the data-name defined as a double-byte alphanumeric data item in the WORKING-STORAGE or LINKAGE section. The input-output execution result is

	Location	Information Type	Details and Use of Specification
			set for this data-name.(*) For detail information, specify a 4-byte alphanumeric data item.

* For information on what value to set, see "Appendix B I-O Status List".

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 data name definitions specified in the file control entry.

Defining Records

Define a record in file and record description entries. The following table lists information required to write a file description entry:

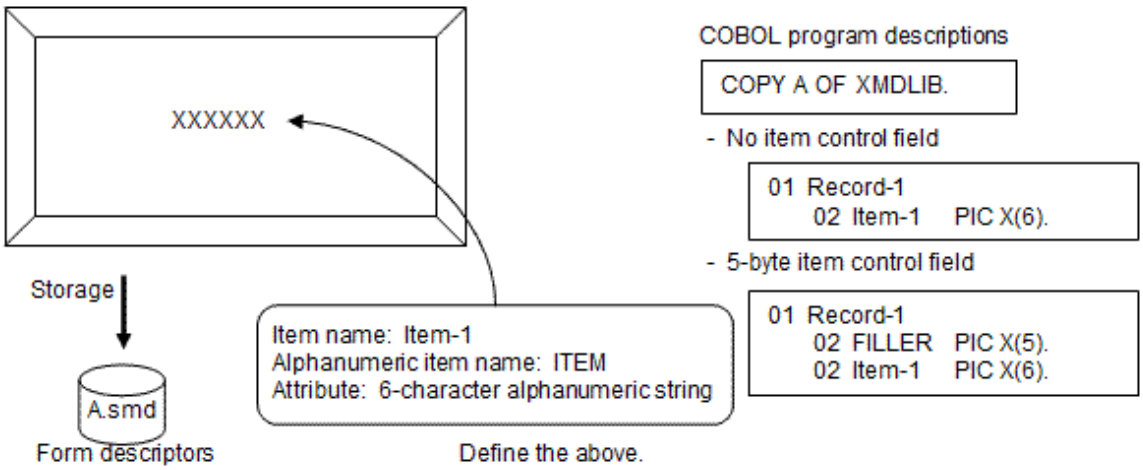
Table 7.12 Information to be specified in a file description entry

	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 for using FORM Overlay Patterns".

Define the partition (item group) defined by the form descriptors in the chart record. This definition statement can be fetched from the form descriptors using the COPY statement with IN/OF XMDLIB is specified in a library at compilation.

The records to be expanded are shown below.



PROCEDURE DIVISION

Execute an input-output statement 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 is defined by form descriptors and the chart record 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 must 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 a chart or a control record in which a form descriptor name is set for the data name specified in the FORMAT clause with the file control entry is output.

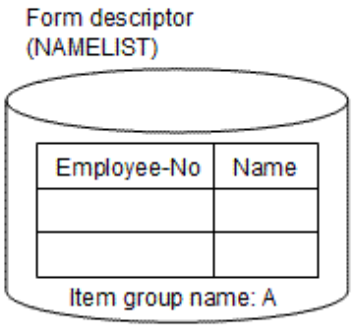
The fixed or irregular form of these pages is maintained on subsequent pages as long as a WRITE statement changing the form of the page is not executed.

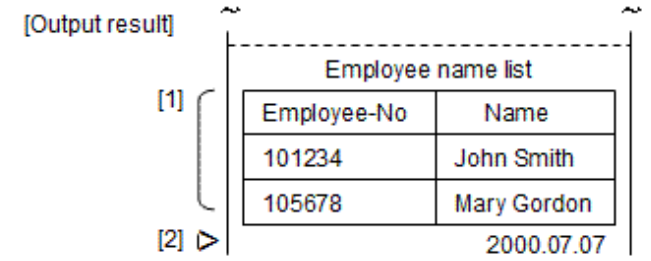
Pages can be changed if changing a form descriptor name that the page format is determined to another name and output. In this instance, AFTER ADVANCING PAGE is specified for a WRITE statement immediately after the output of the control record. See "7.2.2 Program Specifications for using a Print File without FORMAT" for the ADVANCING specification of a WRITE statement.

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.

```
MOVE 101234 TO Employee-No OF chart-record(1).
MOVE 105678 TO Employee-No OF chart-record(2).
MOVE "John Smith"       TO Name OF chart-record(1).
MOVE "Mary Gordon"      TO Name OF chart-record(2).
MOVE "NAMELIST" TO form-descriptor-name-area.
MOVE "A" TO item-group-name-area.
WRITE chart-record AFTER ADVANCING PAGE.  *>[1]
MOVE SPACE TO form-descriptor-name-report-area.
MOVE SPACE TO line-record.
MOVE "2000.07.07" TO print-date OF line-record.
WRITE line-record AFTER ADVANCING 3.          *>[2]
```





Input-Output Error Processing

For information about how to detect I-O errors and execution results when I-O errors occur, refer to "6.6 Input-Output Error Processing".

7.4.3 Program Compilation and Linkage

This section describes program compilation and linkage using form descriptors in a print file.

For compilation and linkage using the cobol command

```
$  cobol -M -m path-name [compiler-and-linkageoptions] file-name
```

Specify the -M option for main programs.

Specify the -m option for the path name of the directory in which a form descriptor is stored.

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.

For linkage using the ld command

For information about how to link using the ld command, refer to "Appendix K ld Command".

7.4.4 Program Execution

To execute a program that uses form descriptors with a print file, a printer information file used by PowerFORM RTS is required.

The following environments must be set to execute a program that uses form descriptors with a print file:

- If specifying a file-identifier in an ASSIGN clause, set the file-identifier as the environment variable with the path name of the printer information file as its value.

  If specifying the file-identifier with relative path name, it is searched for in the following order:

  1. The directory set in the MEFTDIR environment variable.
  2. The current directory.

- If using a form descriptor and a FORM overlay pattern in the print file, set the path name in which the overlay pattern is stored for the printer information file (the setting of the FOVLDIR environment variable is invalid).
- If changing the FCB default value in print files that form descriptors are used, set FCB name used for default in the CBR_FCB_NAME environment variable.
- Set the directory in which PowerFORM RTS is stored in the LD_LIBRARY_PATH environment variable.
- For information about necessary environment setting for other execution, refer to the "PowerFORM Runtime Reference".

Example

Program execution using a form descriptor in a print file

ASSIGN Clause Specification for COBOL Program

```
SELECT  file-name  ASSIGN TO PRTFILE ......
```

Input Command

```
$  LD_LIBRARY_PATH=/home/usr1:$LD_LIBRARY_PATH                    ... [1]
$  export  LD_LIBRARY_PATH                                       ... [2]
$  cobol -M -o PROG1 -m/home/usr1/meddir PROG1.cobol             ... [3]
$  PRTFILE=/home/xx/MEFPRC ; export PRTFILE                      ... [4]
$  PROG1                                                         ... [5]
```

/home/usr1: The directory in which PowerFORM RTS is stored.

/home/usr1/meddir: The directory in which the form descriptor file is stored.

/home/xx/MEFPRC: The printer information file.

PROG1: Executable program.

[1],[2]: Set the directory in which PowerFORM RTS is stored in the LD_LIBRARY_PATH environment variable.

[3]: Compile and link using the Cobol command.

[4]: Set the printer information file used by PowerFORM RTS in the PRTFILE environment variable.

[5]: Execute the executable program. Forms are output to a printer device if the executable program terminates.

Chapter 8 Calling Subprograms (Inter-Program Communication)

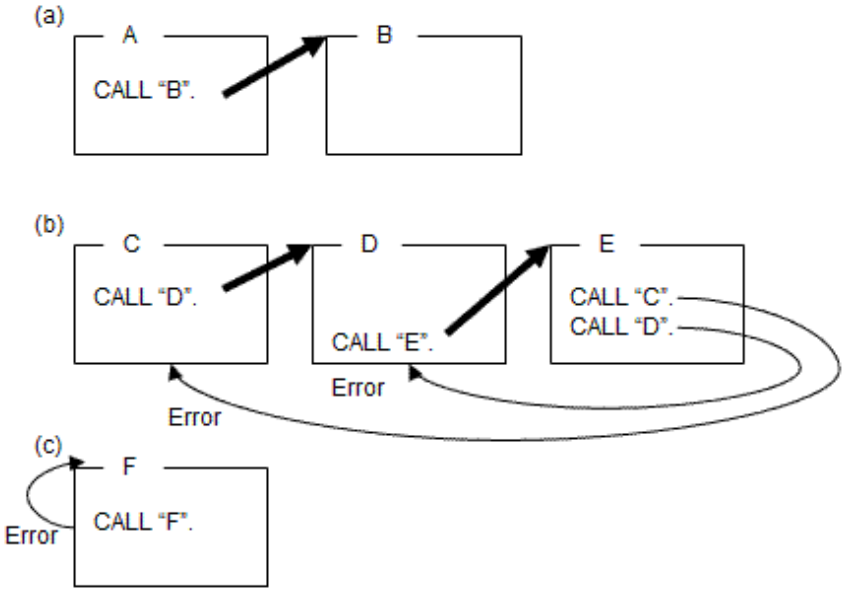
This chapter explains how to call programs from other programs, often referred to as inter-program communication. The following topics are discussed:

8.1 Calling Relationship Types

This section explains types of program calling relationships.

A COBOL program can call other programs and can be called from other programs, as shown in section (a) in Figure below. 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 (b) in Figure below. A COBOL program without a recursive attribute cannot call itself, as shown in section (c) in Figure below.

Figure 8.1 Calling relationship forms



8.1.1 COBOL Inter-Language Environment

This section is an explanation of COBOL runtime environment and run unit.

Runtime Environment and Run Unit

The runtime environment of COBOL programs is generally opened when needed and closed when no longer needed or at the end of a COBOL run unit. However, when multithreaded programs are run, the timing of closing of runtime environment is different. For more details, see "16.3.1 Runtime Environment and Run Unit".

The COBOL run unit refers to the period that starts when control is passed on to the COBOL main program and ends when it returns to the calling side (Figure (a) below) or until the STOP RUN statement is executed (Figure (b) below). An exception is when calling COBOL programs from the programs described in other languages.

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. 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 another language program without calling JMPCINT2, the COBOL program becomes the COBOL main program. Therefore, the opening and closing of the COBOL runtime environment are made for the number of times the COBOL main program is called, which degrades the execution efficiency (Figure (c) below).

However, by calling JMPCINT2, the period until JMPCINT3 is called can be the COBOL run unit and the runtime environment is not closed until JMPCINT3 is called. (Figure (d) below)

For information on how to call JMPCINT2 and JMPCINT3, see "G.2 Subroutines Used in Linkage with another Language".

Processing Performed while the Runtime Environment is Opened or Closed

When the runtime environment is opened, information on runtime initialization files required for COBOL programs to execute is fetched. When the runtime environment is closed, the resources used by COBOL programs are freed. The processing performed while 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 called as shown in (c) in the figure below, the runtime environment of program A and program B differs from that of program C. Therefore, the external data for program A and program B is located in a different area from that for program C. In addition, the caution given below applies to this kind of usage. Therefore, call JMPC1NT2 and JMPC1NT3 from another language program as shown in (d) in the figure below.

Figure 8.2 COBOL Programs Only

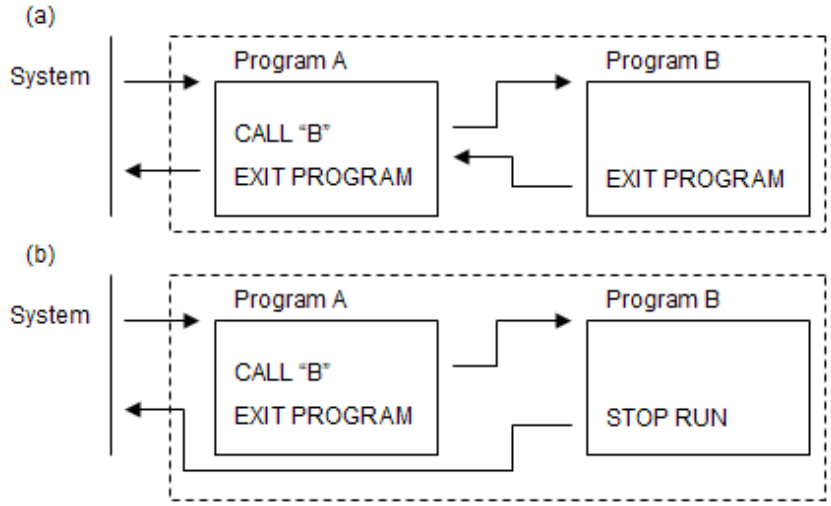
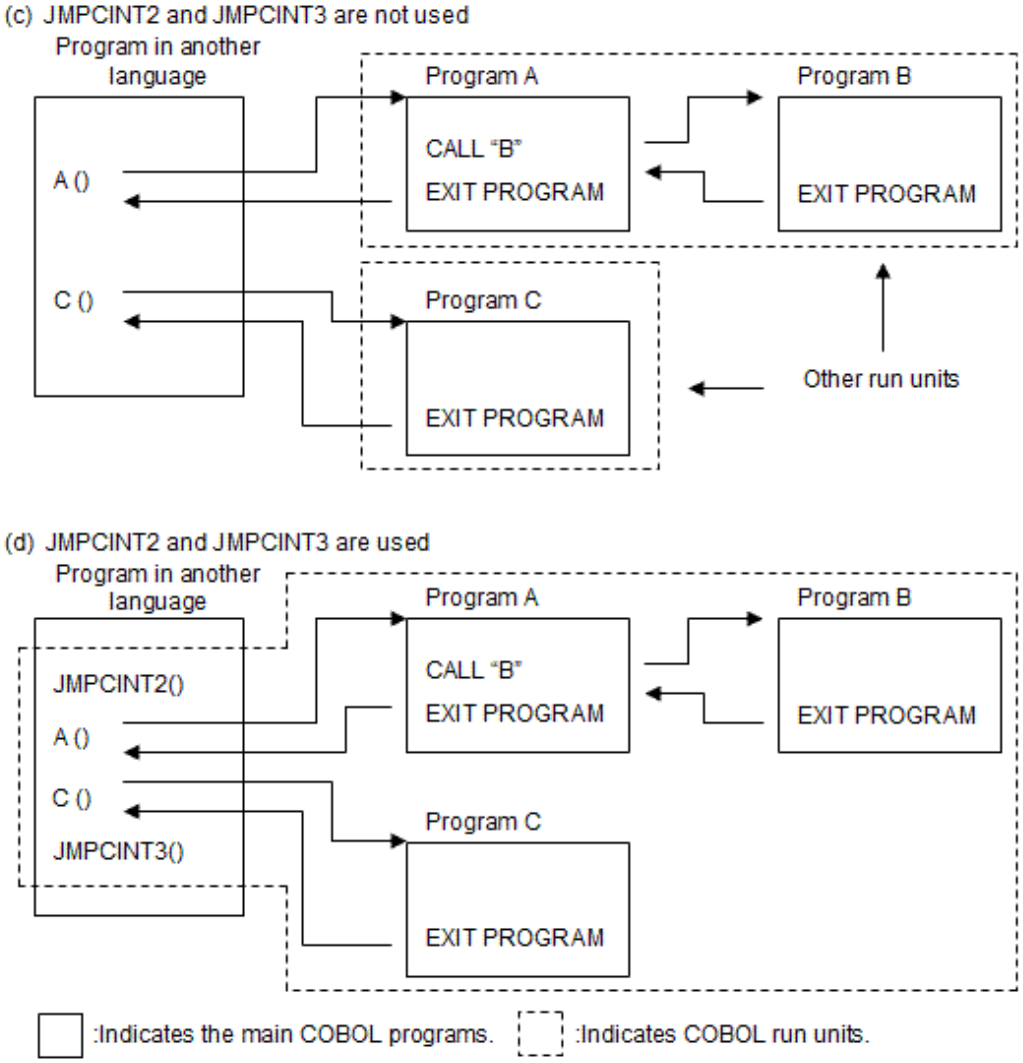


Figure 8.3 Calling COBOL Programs from Other Language Programs



Note

In the following case, do not call the COBOL run unit multiple times from other language program:

- When using external data or external files.
- If unconditional close is made to an opened file.
- If the STOP RUN statement is executed with other than the COBOL main program.
- When using the Object-Oriented programming functions.

8.1.2 Dynamic Program Structure

This section describes the features of dynamic program structure and cautions.

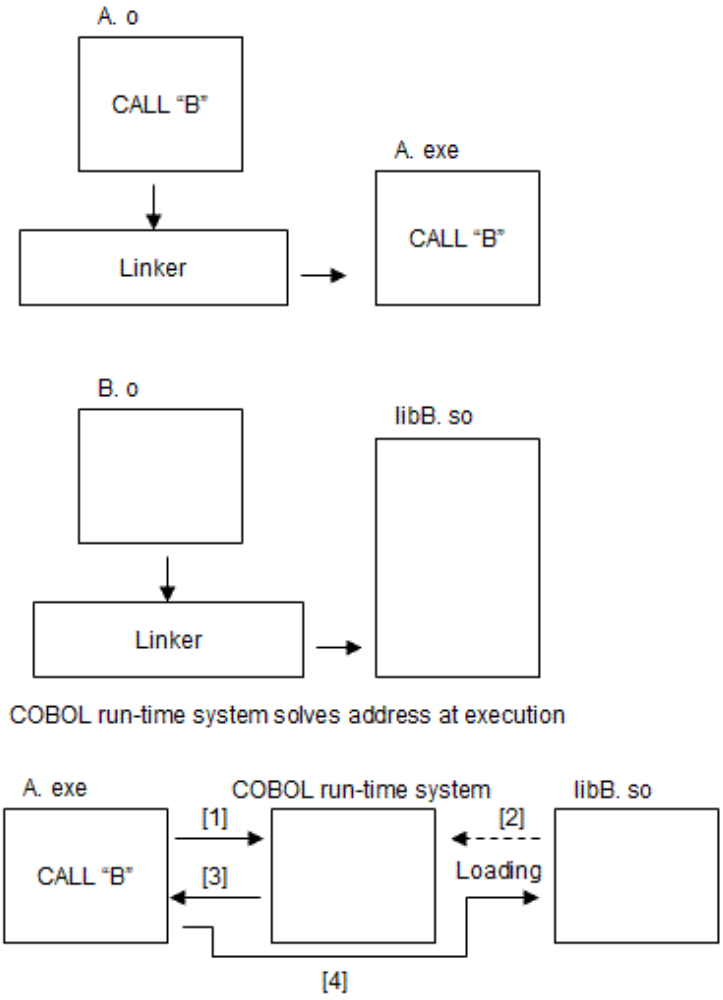
8.1.2.1 Features of Dynamic Program Structure

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 where all the subprograms are loaded at the

startup of the executable file. This simultaneously saves the virtual and actual memory use. Calling of subprograms, however, becomes slower as it is made via the COBOL runtime system.

Performance comparison with dynamic link structure

If calling subprogram with a dynamic link structure, the dynamic linker executes symbol solution when the calling program is called. In this situation, only the symbol solution for subprograms is executed but the subprograms are not loaded in the memories. When the subprogram is actually called, the subprogram is loaded. For this reason, for execution performance when a program of calling side is started, there is very little difference between calling subprograms with the dynamic link structure and dynamic program structure.



Description of the figure:

[1] to [4] shows the processing order.

- [1] The COBOL runtime system is called.
- [2] The COBOL runtime system loads Program B.
- [3] Returns to Program A.
- [4] Branches to Program B.

Subprogram entry information is required for executing program of dynamic program structure. However, entry information is unnecessary if the shared object file name of the subprogram is in the form "libprogram-name.so". For this reason, it is recommended that you create a shared object program for each subprogram that is called using the dynamic program structure, and that you make the file name "libprogram-name.so".

It is important that users of this program configuration fully understand the overall configuration. For details, refer to "8.1.2.3 Cautions".

8.1.2.2 Subprogram Entry Information

Entry information is required if the executable program is dynamic program structure. For details about the entry information format, refer to "4.1.3 Subprogram entry information".

8.1.2.3 Cautions

- Creating and linking subprograms

  When a subprogram is called by dynamic program structure, the subprogram is created as a shared object program. The subprogram and shared object program should correspond one on one according to the rule that the file name of the shared object program is "libsubprogram-name.so".

  If you follow the rule, you do not have to use entry information file or to link subprograms to programs calling the programs or executable programs by using -l option.

  To call more than one subprogram wrapped into one shared object program by dynamic program structure, use the entry information file or link the shared object programs with the executable programs by using -l option. As a result, you can call more than one subprogram wrapped into one shared object program in dynamic program structure. You should be aware, however, that a CANCEL statement for the program called by this method does not have any meaning.

- Program names

  You cannot call program names that use national characters in a dynamic program structure because of a system restriction. Accordingly, you cannot specify the following as CALL statements.

  - Specifying national data items as identifier.
  - Specifying national nonnumeric literal as literal.

- The initial status of programs

  Except the initial programs, 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.

- CALL statements in which an identifier is specified

  - When an identifier is specified in a CALL statement, any characters after the 256 bytes are ignored. This is because the maximum number of the characters that can be an effective program name is 255 bytes from the head of the specified area. Therefore, note that a function name that exceeds 255 bytes cannot be called when you call a C program by the C language collaboration. There is no problem for COBOL because the maximum length of a program name is 160 bytes.
  - If a blank space is included between character strings specified for a CALL statement in which an identifier is specified, characters after the first blank space are ignored.

- Using dynamic program structure and dynamic link structure at the same time in COBOL applications

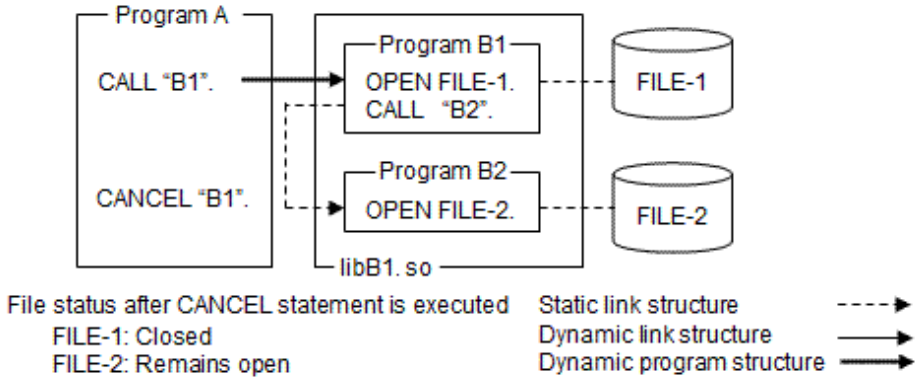
  It is recommended to not use dynamic program structure and dynamic link structure at the same time in COBOL applications because user errors may occur. If you use them, understand each structure enough and note the following:

  - Don't link shared objects of the programs called in an entire COBOL application in common with shared objects of the program called by dynamic program structure. In this case, if the program called using the dynamic program structure is deleted from the memory using the CANCEL statement, programs linked using the dynamic link structure are also deleted from the memory. Thus, the linked programs using the dynamic link structure are initialized, and if they are called from another COBOL application, the intended result might not be achieved.
    Link shared objects of the programs called in common to an entire COBOL application with the executable program by the -l option
  - When the compiler option "DLOAD" is specified, all subprograms called from the program are called by dynamic program structure. Therefore, programs that require being called by dynamic linking structure cannot be called from the program for which "DLOAD" is specified.
  - When a specific program is called by dynamic program structure and dynamic link structure respectively, the program cannot be deleted from memory by the CANCEL statement.

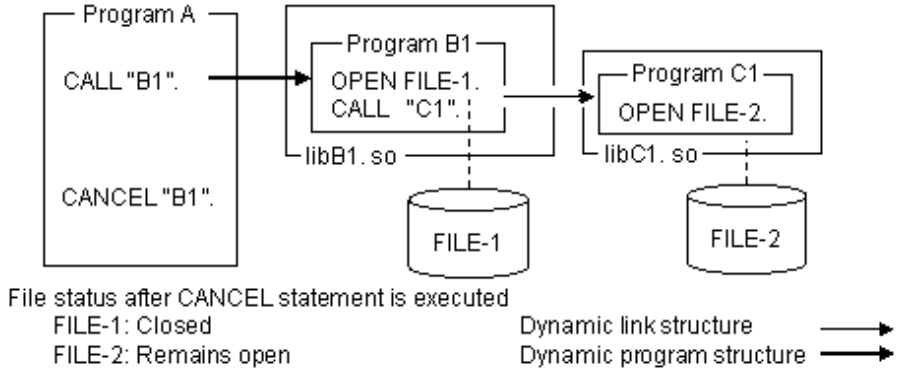
- For the operation after executing the CANCEL statement

  When the CANCEL statement is executed, the subprogram is deleted from the memory. However, if a subprogram which is specified in the CANCEL statement calls a subprogram that is linked using the static link structure or the dynamic link structure, the file that is opened in that subprogram cannot be closed. The behavior in this case cannot be guaranteed. For this reason, make sure that you close the open file before the CANCEL statement is executed.

  <When a subprogram in the simplified structure has been called by the subprogram to be canceled>



  <When a subprogram in the dynamic link structure has been called by the subprogram to be canceled>



Information

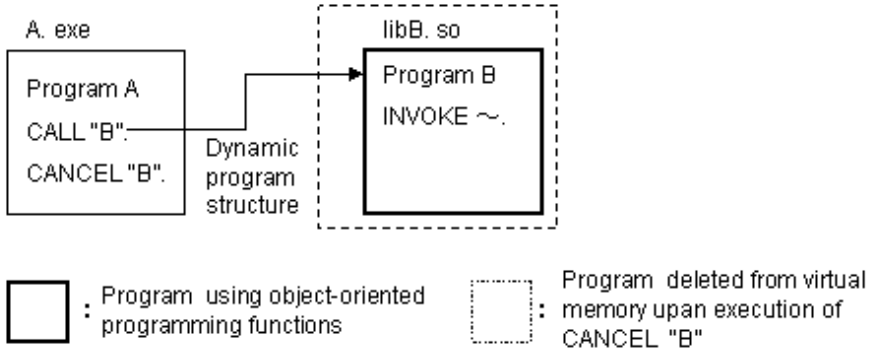
In the above example, libB1.so and libC1.so are loaded when 'CALL "B1" is executed, and libB1.so and libC1.so are deleted from the virtual memory when 'CANCEL "B1"' is executed.

- Do not use CANCEL statement to delete programs that use Object-Oriented programming functions

Example

Example1

Since Program B, which uses Object-Oriented programming functions, is deleted from virtual memory upon execution of 'CANCEL "B"', this CANCEL statement cannot be used.

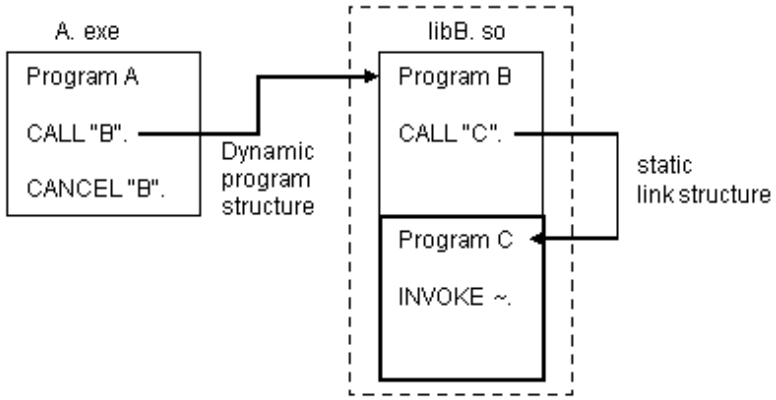


Example 2

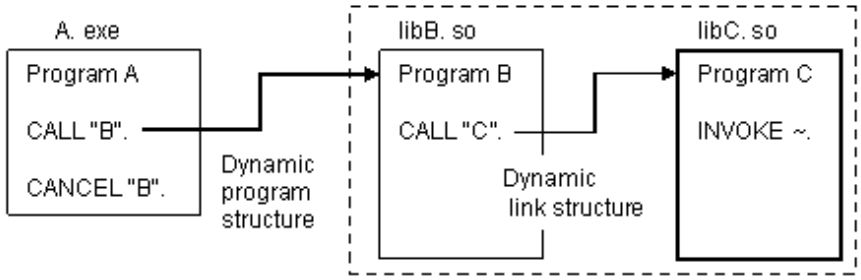
If Program B and Program C using Object-Oriented programming functions are used in the simplified structure or dynamic link structure, Program B and Program C are deleted from virtual memory when 'CANCEL "B"' is executed. So the CANCEL statement is not valid with these program structures (see (a) or (b) in figure below).

However, in this case, the CANCEL statement can be used if Program B and Program C are connected using dynamic program structure as illustrated in diagram (c).

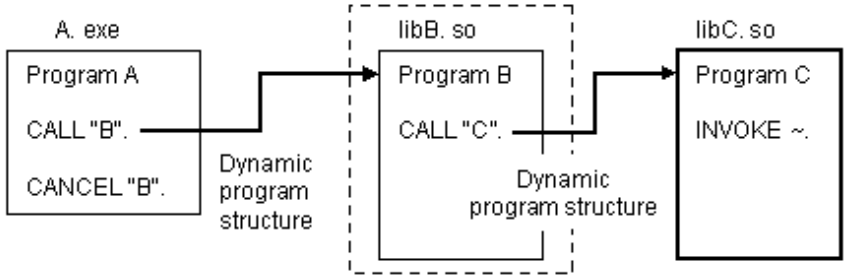
(a) Programs B and Program C are connected using the static link structure

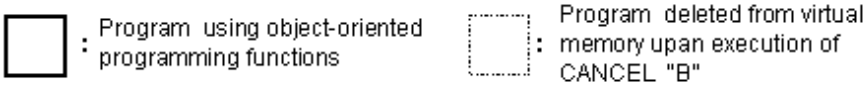


(b) Programs B and Program C are connected using the dynamic link structure



(c) Programs B and Program C are connected using the dynamic program structure





8.2 Calling COBOL Programs from COBOL Programs

This section explains how to call other COBOL programs (subprograms) from COBOL programs (calling programs).

8.2.1 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. However, calling programs by specifying data names in the CALL statement makes the dynamic program structure, regardless of specification of the DLOAD complier option. For details of program structure, refer to "3.2.2 Linkage Type and Program Structure".

8.2.2 Secondary Entry Points

An entry point to call programs can be set in the procedures in COBOL programs. The start point of a program procedure is a primary entry point, and an entry point set in the middle of a procedure is a secondary entry point. When the CALL statement with a program name specified is executed, the subprogram is executed 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, specify an ENTRY statement. When a program is sequentially executed, the ENTRY statement is skipped. An ENTRY statement cannot be specified in internal programs.

8.2.3 Returning Control and Exiting Programs

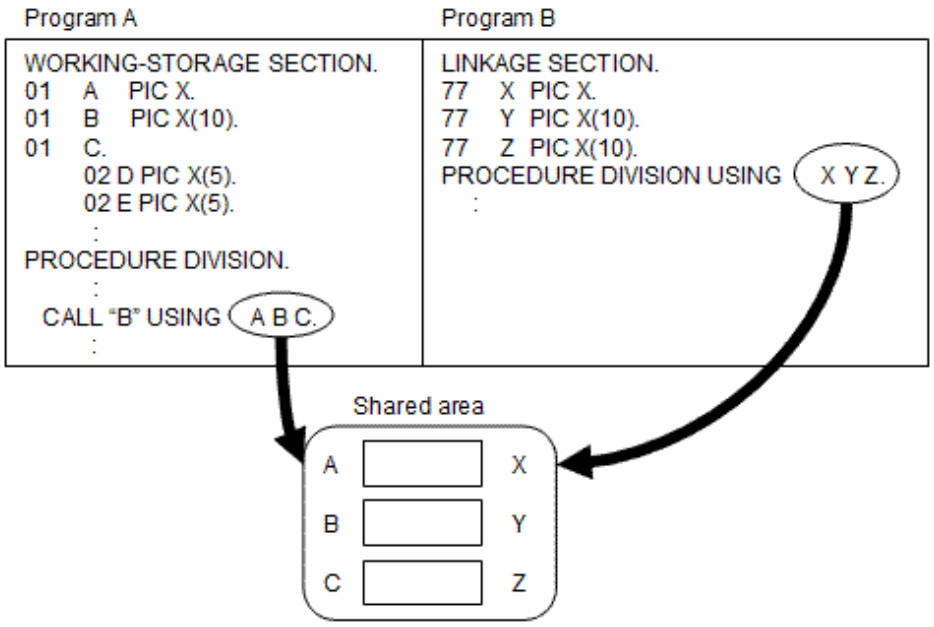
To return control from subprograms to calling programs, execute the EXIT PROGRAM statement. When the EXIT PROGRAM statement is executed, control returns immediately after the CALL statement executed by the calling program. To quit execution of all COBOL programs, execute the STOP RUN statement. When the STOP RUN statement is executed, control returns to the calling source of the COBOL main program.

8.2.4 Passing Parameters

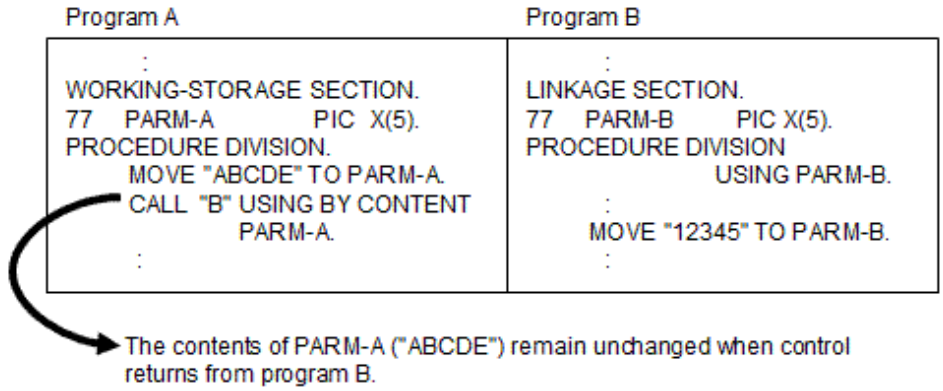
Parameters can be passed between calling programs and subprograms. In a calling program, write data items defined in the FILE, WORKING-STORAGE, or LINKAGE sections in the USING phrase of the CALL statement. In a subprogram, write data-names to receive parameters in the USING phrase of the PROCEDURE DIVISION header or the ENTRY statement.

The order of data-names specified in the USING phrase of the CALL statement of the calling program corresponds to the order of data-names specified in the USING phrase of the PROCEDURE DIVISION header of subprograms or in the USING phrase of the ENTRY statement. 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.



To prevent the contents of parameters of the calling program from being not be changed by the subprogram, specify "BY CONTENT data-name" in the USING phrase of the CALL statement.



Note the following points for receiving parameters by subprograms 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 an ENTRY statement on the subprogram.
- The number of parameters specified in the CALL statement in the calling program matches that written in either the PROCEDURE DIVISION header of a subprogram or the USING phrase of the ENTRY statement. The length of each parameter in the CALL statement also matches the length of each parameter in the PROCEDURE DIVISION header or the USING phrase.
- The number of parameters specified for a CALL statement of a calling program and the number of parameters described in a USING phrase of the PROCEDURE DIVISION header side or of an ENTRY statement on the subprogram must be the same.
- The calling convention specified in a CALL statement of a calling program and the calling convention specified in a PROCEDURE DIVISION header or in an ENTRY statement in subprogram must be 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
The data item for parameter reception is not defined in the linkage section.	Yes	No
A description is missing from the USING phrase of the data item for parameter reception.	Note 1	No
The numbers of parameters or the parameter lengths do not match.	Note 2	Note 2

Note 1 : Descriptions with errors may not be checked if the parameter described in the USING phrase of PROCEDURE DIVISION header and the parameter described in the USING phrase of the ENTRY statement are not the same.

Note 2 : The CHECK compile option must be specified at the program compile time. The CALL statement that calls an internal program is checked at compile time, and the CALL statement that calls an external program is checked at execution time. See "5.3 Using the CHECK Function"

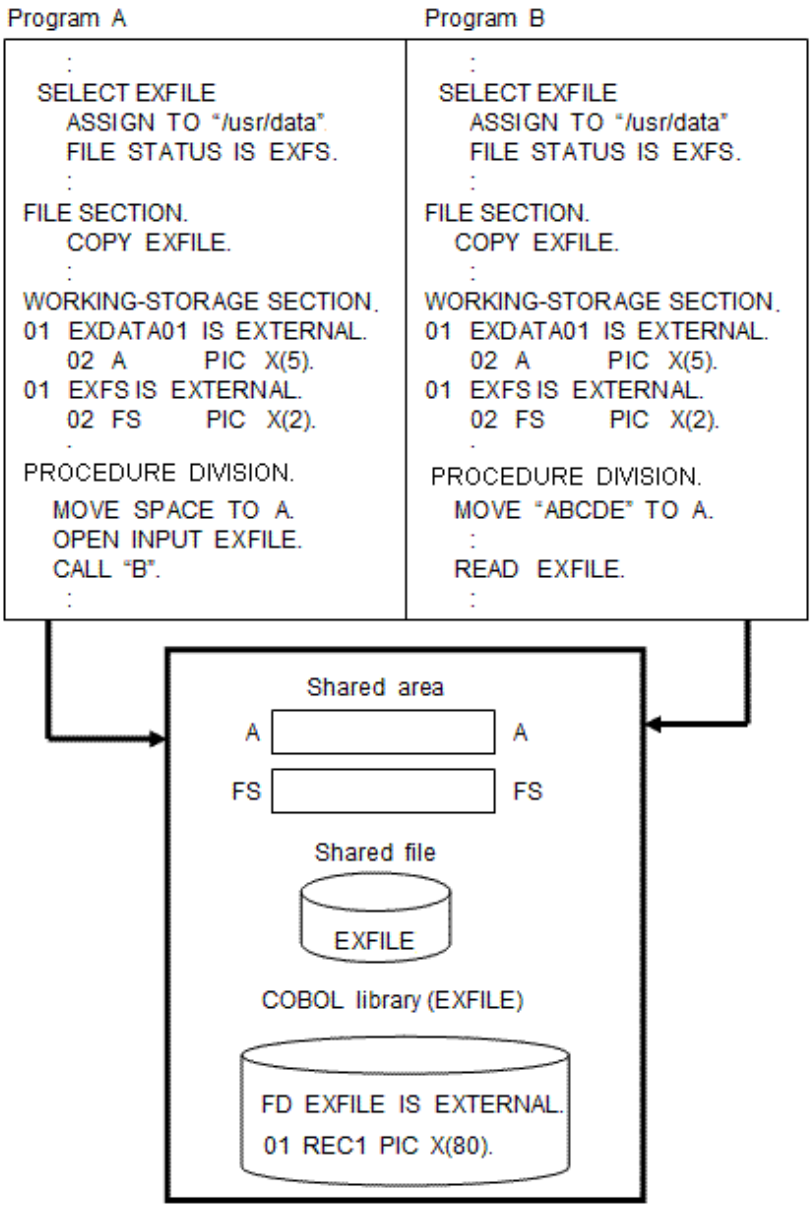
Note

- When a COBOL program is called from another COBOL program, "USING BY VALUE" cannot be specified.
- When an object reference data item is specified as parameters, the USAGE clause of the object reference data items should be the same.

8.2.5 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.

Program maintenance ability can be improved by generating the definitions of external data or files as a COBOL library, then including the data in programs with the COPY statement.



8.2.5.1 Notes on n 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 ends. That is, the external data area cannot be freed even if the program is deleted by the CANCEL statement. Be careful when external data is used in a program that is called repeatedly.

8.2.5.2 Notes on Using an External File

- An external file can be shared by more than one program. 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 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 "NetCOBOL Language Reference".
- Since many attribute items of an external file must be the same, it is recommended to use a COBOL library wherever possible. Since these items are checked at the runtime, 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 ends. (These areas of a 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.

8.2.6 Return Codes

When control is returned to calling programs from subprograms, return code values can be passed either using the RETURNING phrase or the special register PROGRAM-STATUS (or RETURN-CODE).

The RETURNING phrase uses data items defined by users to transfer return code values.

You specify the RETURNING phrase in the CALL statement of the program to be called and in the PROCEDURE DIVISION header 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 occurred
      THEN
        MOVE 99 TO RTN-CD
      ELSE
        MOVE  0 TO RTN-CD
      END-IF.
```

Note

If the RETURNING phrase is specified in the CALL statement, the value of the special register PROGRAM-STATUS in the calling program remains unchanged. Also, if RETURNING phrase is specified in the CALL statement, the subprogram has to set a value in the data item specified in the RETURNING phrase. If no value is set, the value of the data item specified in the RETURNING phrase in the calling program is undefined.

The PROGRAM-STATUS special register is implicitly declared as "PIC S9(18) COMP-5" and the user does not need to define this register in the program.

If a subprogram sets a value in the PROGRAM-STATUS special register, that value is also set in the PROGRAM-STATUS special register of the calling program.

PROGRAM-A

```
  :
MOVE 0 TO PROGRAM-STATUS.
CALL "B".
IF PROGRAM-STATUS NOT = 0
  THEN
  :
```

PROGRAM-B

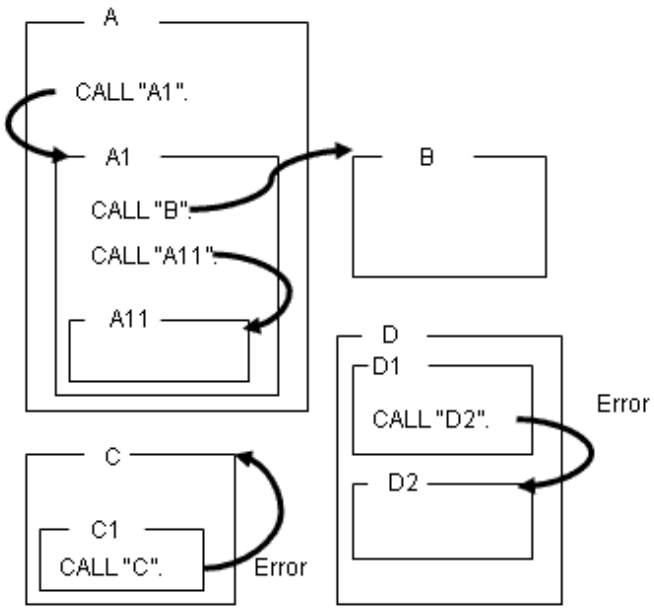
```
  :
IF  error occurred
  THEN
    MOVE 99 TO PROGRAM-STATUS
    EXIT PROGRAM
  END-IF.
```

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.

8.2.7 Internal Programs

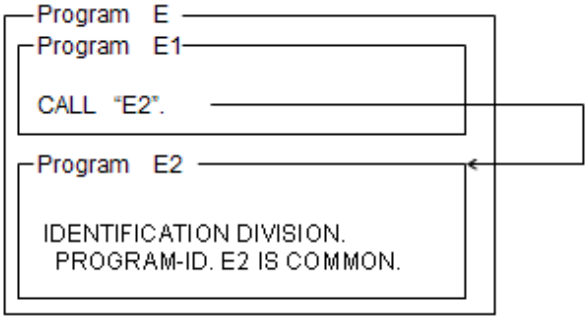
COBOL programs are classified into external and internal programs based on the program structure. The outermost program not included in other programs is an external program. Programs directly or indirectly included in an external program are internal programs.

An external program (A) can call its internal program (A1). The internal program (A1) can call other external programs (B) or its internal program (A11). However, internal programs cannot call outside programs other than common programs (C from C1, D2 from D1).



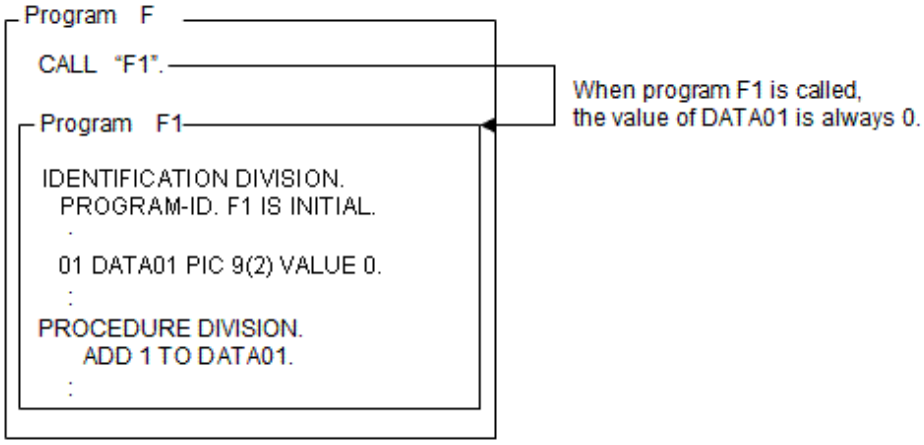
Common Programs

To call an internal program from an outside internal program, specify COMMON in the PROGRAM-ID paragraph of the called internal program. A Program with COMMON specified is common program, and it can be called from internal programs not including containing it.



Initial Programs

To place a program in the initial state whenever it is called, specify INITIAL 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.



Valid Scope of Names

To use data items (defined in the outer programs) in the internal programs, specify the GLOBAL clause in the data description entry. Normally, names are valid only within the program, but data items with the GLOBAL clause specified can be used in internal programs. Outer programs cannot use data items with GLOBAL clause specified in the internal programs.

8.2.8 Notes

- If the compile option ALPHAL is used for both 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 specified in the PROGRAM-ID paragraph are treated as uppercase.

  For more information, see "A.2.1 ALPHAL (lowercase handling (in the program))".
- When compiling the calling program and subprogram, specify the same compile option, ALPHAL or NOALPHAL for both programs. Specify compile option NOALPHAL when you want lowercase letters to be distinguished in your program names.
- When compiling a main program, compile option MAIN must be specified. When compiling subprograms, compile option NOMAIN must be specified. See "A.2.23 MAIN (main program/sub-program specification)".
- Because of system restrictions, program names and method names consisting of national language characters cannot be called in the dynamic link structure or dynamic program structure. Therefore, the following cannot be specified in the CALL or INVOKE statement:
  - National language item names for the identifier.
  - National language character literal for the literal.
- Correct operation is not guaranteed if more than one program with the same program name exists within a COBOL execution unit. The following points detail what will happen if a program with the same name is called during execution:
  - If it is the same program name but the COBOL source program is different, message JMP0032I-U is output at execution.
  - If it is the same COBOL source program but the compile date is different, message JMP0032I-U is output at execution.
  - If the same program is included in more than one .so file, the status called previously cannot be retained and an unintended operation may be performed.

8.3 Linkage with C Language Program

This section explains how to call C programs (functions) from COBOL programs, and how to call COBOL programs from C programs. In this section, programs described in C language are simply called C programs.

8.3.1 Calling C Programs from COBOL Programs

This section explains how to call C programs from COBOL programs.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. PROGRAM-NAME.
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 DATA-1.
  02 ELEMENT-1  PIC 9(4).
  02 ELEMENT-2  PIC X(10).
 01 DATA-2      PIC S9(4) COMP-5.
 01 DATA-3      PIC X(1).
PROCEDURE DIVISION.
  CALL FUNCTION-NAME USING DATA-1 DATA-2 BY VALUE DATA-3.
  IF PROGRAM-STATUS ...
END PROGRAM PROGRAM-NAME.
```

```
typedef struct
{
 char element1[4];
 char element2[10];
}  structure_name;
long int function_name( structure_name *argument1,
                        short int      *argument2,
                        char           *argument3 )
{
    ...
 return(function_value);
}
```

8.3.1.1 Calling Method

To call C programs from COBOL programs, specify function names in the CALL statement of COBOL program. When the return statement is executed in the called C program, control returns immediately after the COBOL CALL statement.

8.3.1.2 Passing Parameters

To pass parameters from COBOL programs to C programs, specify data-names in the USING phrase in the CALL statement. The parameters to be passed to the C program are the addresses or the values of the data items. The relationship between the description of the USING phrase and the contents of parameters is explained below.

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 (for correspondence between COBOL and C data types, see "Table 8.1 Correspondence between COBOL data items and C data types") corresponding to the attribute of the parameter to be passed as a dummy argument in the C program.

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.

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 phrase

This section explains the differences between the BY REFERENCE and BY CONTENT phrases, and between BY REFERENCE and BY VALUE phrases, using program samples.

Difference between BY REFERENCE and BY CONTENT phrases

Source program

```
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" USING BY REFERENCE PRM1 BY CONTENT PRM2.
     DISPLAY "PRM1=" PRM1.
     DISPLAY "PRM2=" PRM2.
 END PROGRAM MAINCOB.
```

```
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 phrase is updated to 20, while the content of PRM2 with the BY CONTENT phrase 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 phrases

Source program

```
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" USING BY REFERENCE PRM1 BY VALUE PRM2.
     DISPLAY "PRM1=" PRM1.
     DISPLAY "PRM2=" PRM2.
```

```
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 phrase, the BY VALUE phrase 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.

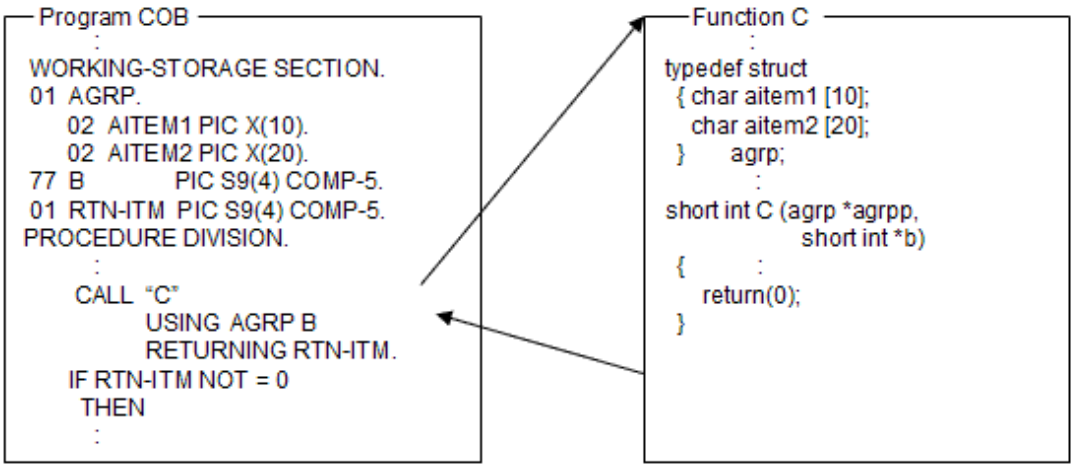
Information

If the BY phrase is omitted, BY REFERENCE is assumed.

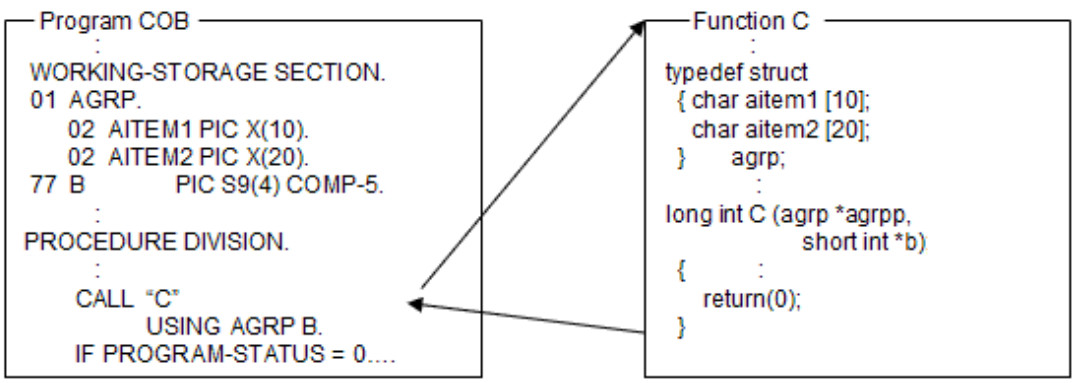
8.3.1.3 Return Codes (Function Values)

Use the special register PROGRAM-STATUS or RETURNING phrase in the call statement to receive return codes (function values) from C programs.

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. Refer to "8.3.3 Correspondence of COBOL and C Data Types" for more information.



If you want to use the special register PROGRAM-STATUS, C function values must be of the long int type. For example:



Note

- When a C program of apart from long int type is called, you can receive the C function values specify RETURNING in CALL statement.

  Example : short int type C program call

```
*>       :
01  function-value    PIC S9(4) COMP-5.
*>       :
     CALL "Cprog" RETURNING function-value.
     IF function-value = 0 THEN
*>       :
```

  [Remarks]

  The attributes of special register PROGRAM-STATUS corresponds to the long int type of C. Consequently, if a C program of short int type is called, referring to the value of the special register PROGRAM-STATUS, without redefinition as shown above, may not provide the correct function value.

- When you call void type C programs, the special register PROGRAM-STATUS is updated to an invalid value. To prevent the special register PROGRAM-STATUS being updated, describe a dummy data item (PIC S9(9) COMP-5) in RETURNING as shown in the example

  Example : void type C program call

```
*>       :
01  dummy-ret    PIC S9(9) COMP-5.
*>       :
     CALL "Cprog" RETURNING dummy-ret.
*>       :
```

8.3.2 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(),JMPCINT3();
 extern long int  COBSUB(structure_name *, short int *);
 structure_name  actual_argument1;
 short int       actual_argument2;
  JMPCINT2();
```

```
  if (COBSUB(&actual_argument1, &actual_argument2))
  {
     // :
  }
  JMPCINT3();
  return(0);
}
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. COBSUB.
 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 COBSUB.
```

8.3.2.1 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.

If the main program is a C program that calls COBOL programs, call JMPCINT2 before calling the first COBOL program. Also call JMPCINT3 after calling the last COBOL program. JMPCINT2 is a subroutine to proceed to initialize COBOL programs. JMPCINT3 is a subroutine to proceed to terminate COBOL programs.

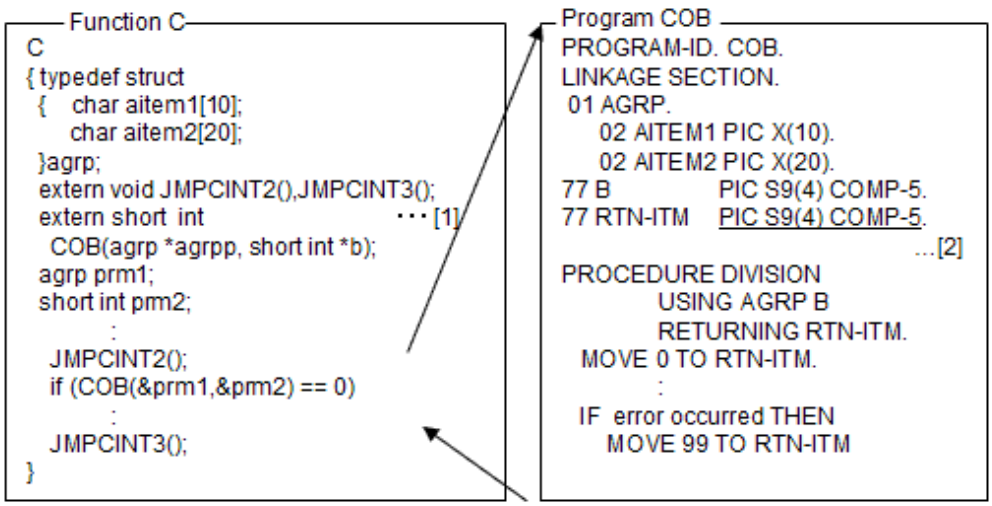
If calling COBOL programs without calling JMPCINT2 and JMPCINT3, the open/close processing for runtime environment of COBOL programs is executed for every COBOL program call. Therefore, execution performances degrade.

8.3.2.2 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 programs specify data-names in the USING phrase of the PROCEDURE DIVISION header or the 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.

8.3.2.3 Return Codes (Function Values)

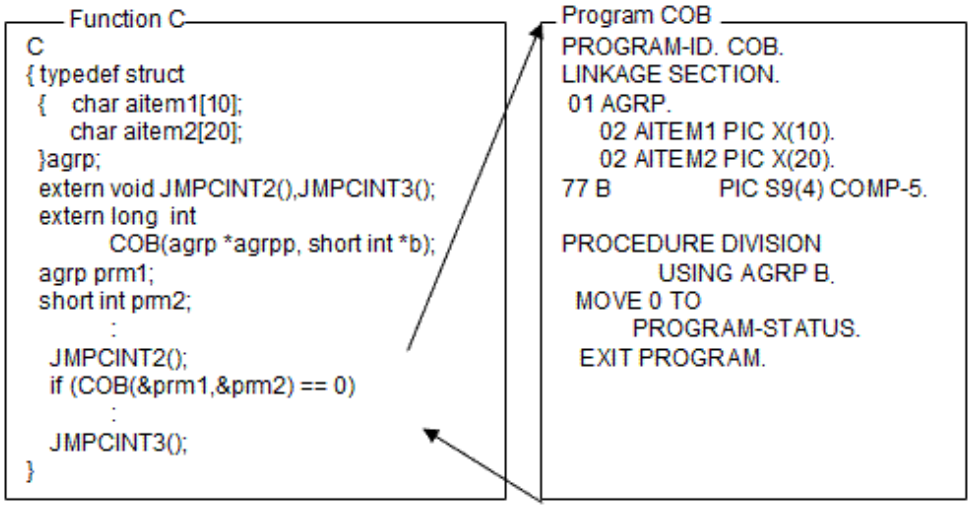
Items in the PROCEDURE DIVISION with RETURNING phrase and special register PROGRAM-STATUS are passed to C programs as function values. You need to ensure that the data types of RETURNING phrase items correspond to the C data types (refer to [1] and [2] of the following figure) used. For details about correspondence of data types, refer to "8.3.3 Correspondence of COBOL and C Data Types".



Note

If describing the RETURNING phrase in the PROCEDURE DIVISION header, the value set in the special register PROGRAM-STATUS is not passed to the C program called.

If you want to use the PROGRAM-STATUS, function values must be of the long int type. For example:



8.3.3 Correspondence of COBOL and C Data Types

The combinations of the attributions for data passed between COBOL and C programs are options. However, "Table 8.1 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

When using COBOL internal Boolean data items and C bit fields, note the storage area locations. Refer to the NetCOBOL Language Reference for information about the storage area locations of COBOL internal Boolean data items. Refer to the C language manuals for the storage area locations of C bit fields.

Table 8.1 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
Zoned 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)(*5)		unsigned char	01 E USAGE IS BINARY-CHAR UNSIGNED	unsigned char E;	1 byte
		short int	01 E PIC S9(4) COMP-5. or 01 F USAGE IS BINARY-SHORT SIGNED.	short int F;	2 bytes
		int	77 G PIC S9(9) COMP-5. or 77 G USAGE IS BINARY-LONG SIGNED.	int G;	4 bytes
		long int	77 H PIC S9(18) COMP-5. or 77 H USAGE IS BINARY-DOUBLE SIGNED.	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	float	01 J COMP-1.	float J;	4 bytes
	Double-precision	Double	01 K COMP-2.	double K;	8 bytes

- *1 :

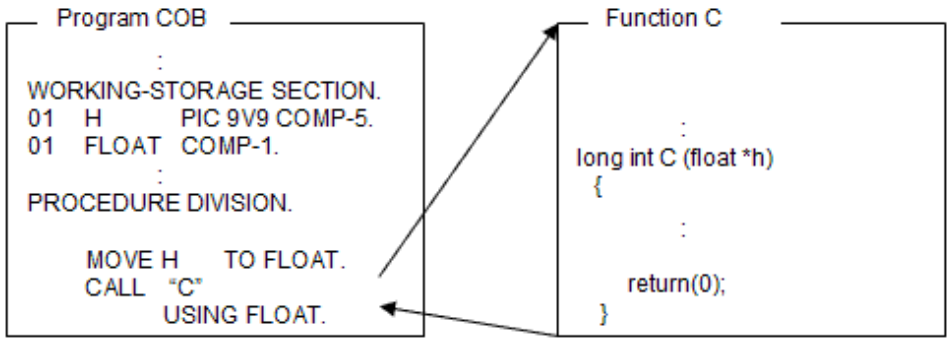
  The internal format of a zoned decimal item in COBOL is a character string consisting of a character representing a sign and numeric characters. C programs treat the item as character data, not numeric data. C programs must perform a type conversion in order to treat zoned decimal items as numeric data.

- *2 :

  The USAGE IS COMP-5 binary item corresponds to short int, int, or long int of a C program, depending on the number of digits as follows:

  - 1 to 4 digits (3 to 4 digits if the BINARY(BYTE) option is specified): short int
  - 5 to 9 digits (7 to 9 digits if the BINARY(BYTE) option is specified): int
  - 10 to 18 digits (17 to 18 digits if the BINARY(BYTE) option is specified): long int

  If the binary item has a decimal part, transfer it via a floating point as follows:



- *3 :

  If the short int, int or long int value of the C program received for the USAGE IS COMP-5 item exceeds the number of digits in the PICTURE clause, the intended result may be unobtainable in subsequent processing. In this case, use the USAGE IS BINARY-SHORT SIGNED, USAGE IS BINARY-LONG SIGNED, or USAGE IS BINARY-DOUBLE SIGNED item.

- *4 :

  When declaring group items as a structure, note the storage boundary of members included in the structure. For details about alignment of the data item storage boundary with COBOL, refer to the "NetCOBOL Language Reference". For C variable storage boundary alignment, refer to C language manuals.

- *5 :

  When receiving the address in the area from a C program, use the pointer data item.

8.3.4 Sharing Data with C Language Programs

Data with the same name can be shared between an external data item of a COBOL program and an external variable of a C program. To provide the external data item of the COBOL program with the attribute that enables sharing with the external variable of the C program, specify the EXTERNAL clause.

- You can refer to external data items of COBOL as external variables of the C language program by specifying DEFINITION (Or DEF).
- You can refer to external variables of the C language program as an external data item of COBOL by specifying REFERENCE (Or REF).

This enables definitions of external data item with shared attributes and references to exist in a program.

EXTERNAL { DEFINITION / DEF }

Specify when the external data item that can be referred from a C language program is defined (COBOL programs have the substance of data).

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.  program-name.

DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  data1
     PIC X(10) EXTERNAL DEFINITION.
 01  data2
     PIC S9(4) COMP-5 EXTERNAL DEF.
PROCEDURE DIVISION.
    CALL "FunctionName".
    ~

END PROGRAM program-name.
```

```
extern char       data1[10];
extern short int  data2;
long int   FunctionName ( )
  {
    ~
  return (0);
  }
```

EXTERNAL { REFERENCE | REF }

Specify when referring to the external variables defined by the C language program (the C language program has the substance of data).

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.  program-name.

DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  data1
     PIC X(10) EXTERNAL REFERENCE.
 01  data2
     PIC S9(4) COMP-5 EXTERNAL REF.
PROCEDURE DIVISION.
   CALL   "FunctionName".
   ~
END PROGRAM program-name.
```

```
char       data1[10];
short int  data2;
long int  FunctionName ( )
  {
    ~
  return (0);
  }
```

Note

In a run unit, specify different data names between external data items with a common attribute to external variables of the C language program and those without~it.

8.3.5 Compiling and Linking Programs

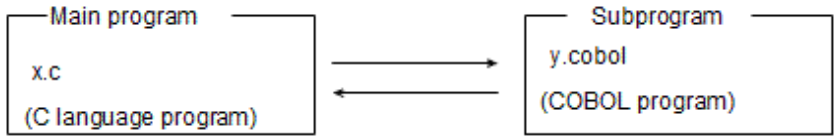
This section explains how to compile and link applications with a format that COBOL programs are called from C language programs using examples for each program structure.

For details about program structures, refer to "3.2.2 Linkage Type and Program Structure".

Note

For linking with other language programs such as C++, the libraries needed to execute the other language programs must be linked. For information about the libraries needed to execute the other language programs, refer to the manuals of the other language.

Example



For simplified structure

```
$  gcc -c -o x.o x.c              [1]
$  cobol -dn -o x x.o y.cobol     [2]      (*)
```

- [1] Generates the relocatable program (x.o).
- [2] Generates the executable program (x).

* Link the libraries that the other language explicitly or implicitly uses appropriately.

For dynamic link structure

```
$  cobol -dy -shared -o liby.so y.cobol  [1]
$  gcc -c -o x.o x.c                     [2]
$  cobol -dy  -ly -L. -o x x.o           [3]      (*)
```

- [1] Generates the shared object (liby.so).
- [2] Generates the relocatable program (x.o).
- [3] Generates the executable program (x).

* Link the libraries that the other language explicitly or implicitly use appropriately.

For dynamic program structure

Dynamic program structure can be realized with a format that COBOL programs are called from C language programs. Loading, calling, and deleting COBOL programs using the dlopen/dlsym/dlclose function from C language programs is required.

In addition, if calling JMPCINT3, deletion of COBOL programs by the dlclose function is required after calling JMPCINT3.

Sample Programs

x.c (C language program)

```
 #include <stdio.h>
 #include <dlfcn.h>

 main()
   {
   void * handle;
   long int (*fptr)(char *);
   char *arg,*errp;
   long int rtncd;

   arg="SAMPLE PROGRAM";

/* Loading the COBOL subprogram into a virtual memory space */
   handle=dlopen("liby.so", RTLD_NOW);

    /* Judging whether loading was successful or unsuccessful */
     if (handle==NULL) {
```

```
      /* Fetching diagnostic information */
      errp=dlerror();
      printf("%s\n",errp);
      return 1;
    }
   /* Fetching the entry point 'y' address of the COBOL subprogram */
    fptr=(long int(*)(char *))dlsym(handle,"y");

   /* Calling the COBOL subprogram */
    rtncd=(*fptr)(arg);

   /* Deleting the COBOL subprogram from the virtual memory space */
    dlclose(handle);
  }
```

y.cobol (COBOL program)

```
      @OPTIONS NOALPHAL
IDENTIFICATION  DIVISION.
PROGRAM-ID.     y.
ENVIRONMENT     DIVISION.
DATA            DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 PARM  PIC X(30).
PROCEDURE       DIVISION
                USING PARM.
    DISPLAY PARM.
    EXIT PROGRAM.
```

```
$  cobol -dy -shared -o liby.so y.cobol [1]
$  gcc -c -o x.o x.c                    [2]
$  cobol -dy -o x x.o  (See Note)       [3]      (*1)(*2)
```

- [1] Generates the shared object (liby.so).
- [2] Generates the relocatable program (x.o).
- [3] Generates the executable program (x).

*1 Since subprograms are loaded using the dlopen function in a main program in the dynamic link structure, linking the subprograms is not required when generating executable programs.

*2 Link libraries that the other language explicitly or implicitly use appropriately.

8.3.6 Notes

This section describes points to be aware of when programs are executed.

- Unlike C character strings, no null characters are inserted automatically at the end of COBOL character strings.
- 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.
- 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 when programs consisting of lower case characters are called by the CALL statement with literal specifications.

  For example the statements:

```
CALL "abc".
```

will Call the program "abc" when NOALPHAL is specified. But if the ALPHAL, is specified the program "ABC" will be called.

CALL statements that use a data name are not affected by the ALPHAL option. Thus:

```
MOVE "abc" TO A.
CALL A.
```

Calls program "abc" regardless of whether option ALPHAL or NOALPHAL is specified.

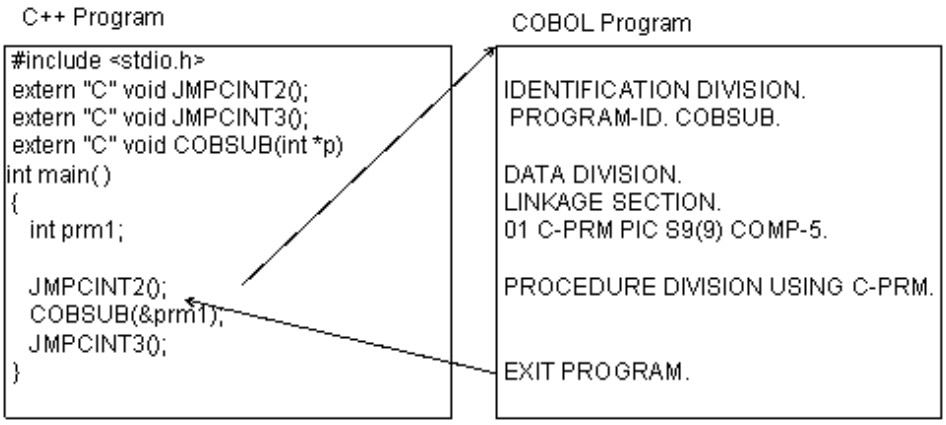
C program names can be case sensitive so use the NOALPHAL COBOL compile option to prevent the compiler converting all program names to upper case for program names consisting lower case.

For example:

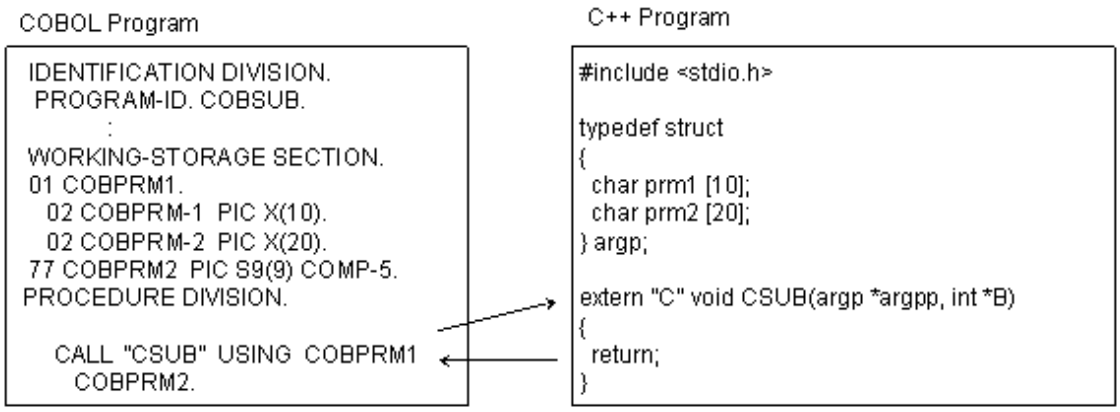
```
PROGRAM-ID. abc.
```

When NOALPHAL is specified, the program name is "abc". When ALPHAL is specified, the program name is "ABC".

- If calling C programs from COBOL programs, data items used for parameters must be boundary of C variables. For details about C variable storage boundary alignment, refer to C language manuals.
- Even if the main program is another language program and COBOL program is called, libraries needed to execute a COBOL program must be linked to the main program. For information about libraries, "Appendix K ld Command".
- To call a COBOL program from a C++ program (the file extension is cpp or cxx), or to call a C++ program from a COBOL program, specify "extern "C" as shown below.
  - Calling a COBOL program from a C++ program



  - Calling a C++ program from a COBOL program



Chapter 9 Using ACCEPT and DISPLAY Statements

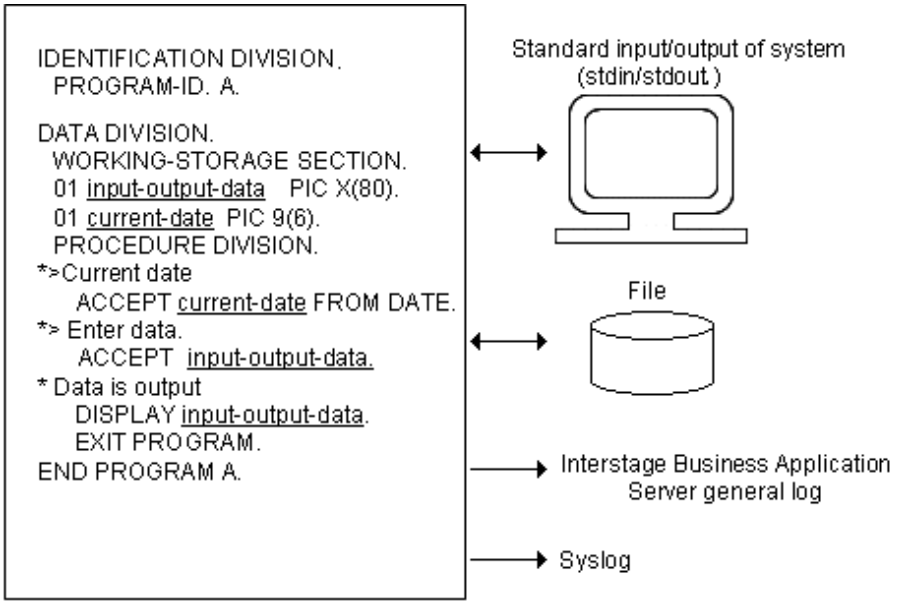
This chapter offers tips on using the ACCEPT and DISPLAY statements, including I-O destination types and specification methods, using standard input (stdin), standard output (stdout), standard error (stderr), program using files, and entering the current date and time. Additionally, it describes fetching command line arguments and environment variable handling.

9.1 ACCEPT/DISPLAY Function

This section describes the ACCEPT/DISPLAY function that sends and receives data using the ACCEPT and DISPLAY statements. Sample programs using this function are provided for your reference.

9.1.1 Outline

With the ACCEPT/DISPLAY function, data can be input and output easily with a system standard input-output (stdin/stdout), the Interstage Business Application Server general log, Syslog, or files. Additionally, the current date and time can be read from the system.



9.1.2 Input/Output Destination Types and Specification Methods

The input/output destination of data depends on:

- ACCEPT statement FROM phrase.
- DISPLAY statement UPON phrase.
- Compiler option specifications.

The Table below lists the relationship between these specifications and input/output destinations:

Table 9.1 Input/output destinations of the ACCEPT/DISPLAY function

	FROM or UPON Specification	Compiler Option to be Specified	Input/Output Destination
1	None or mnemonic-name corresponding to the function-name SYSIN/ SYSOUT.	None	- System standard input-output (stdin/stdout). - Syslog if the CBR_DISPLAY_SYSOUT_OUTPUT environment variable is specified. (*6)(*7) - If the CBR_COMPOSER_SYSOUT environment variable is specified, this complies with the settings for the

	FROM or UPON Specification	Compiler Option to be Specified	Input/Output Destination
			management name defined by log definition file specified on the right side. (*3)(*4)
		SSIN (environment variable name). SSOUT (environment variable name).	- Files set in the environment variable name at program execution. (*1) - Syslog if the CBR_DISPLAY_SYSOUT_OUTPUT environment variable is specified. (*6)(*7) - If the CBR_COMPOSER_SYSOUT environment variable is specified, this complies with the settings for the management name defined by log definition file specified on the right side. (*3)(*4)
2	Mnemonic-name corresponding to the function-name SYSERR.	None	- System standard error (stderr). - Specified file if the CBR_MESSOUTFILE environment variable is specified. - Syslog if the CBR_DISPLAY_SYSERR_OUTPUT environment variable is specified. (*6)(*7) - If CBR_COMPOSER_SYSERR environment variable is specified, this complies with the settings for the management name defined by log definition file specified on the right side. (*4)
3	Mnemonic-name corresponding to the function-name CONSOLE.	None	- System standard input-output (stdin/stdout) (*2) - Syslog if the CBR_DISPLAY_CONSOLE_OUTPUT environment variable is specified. (*6)(*7) - If the CBR_COMPOSER_CONSOLE environment variable is specified, this complies with the settings for the management name defined by log definition file specified on the right side. (*3)(*5)

*1 : If SYSIN/SYSOUT is specified as the environment variable name in the SSIN/SSOUT compile option, the system standard input-output is the input-output destination.

*2 : Usually, item 1 is applicable when the system standard input-output is the input-output destination.

*3 : Data cannot be input to the Interstage Business Application Server general log.

*4 : If data is output to the Interstage Business Application Server general log, the specification of another output destination is invalid.

*5 : If data is output to the Interstage Business Application Server general log, no data is output to the standard output (stdout).

*6 : Syslog cannot log data input.

*7 : When Syslog is specified for output, no other specified destinations will receive output. If the Interstage Business Application Server general log is specified for output along with Syslog, no output is sent to Syslog.

Both 1 and 3 cannot be used for entire programs operated from the start to end of execution. The first ACCEPT statement executed or the description specified in the DISPLAY statement is validated in programs.

9.1.3 Programs Using System Standard Input-Output (stdin/stdout)

This section explains how to write, compile, link, and execute programs, and provides an example of the simplest description for programs using a system standard input-output (stdin/stdout).

9.1.3.1 Program Specifications

This section explains the program descriptions with system standard input-output (stdin/stdout) for each COBOL division.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  data-name  ... .
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 set output data.

PROCEDURE DIVISION

Use an ACCEPT statement to input data from a standard input. 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 until the entered data satisfies the specified length. Use a DISPLAY statement to output data to the standard output. 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, the specified character string is output.

9.1.3.2 Program Compilation and Linkage

Do not specify the compiler options SSIN and SSOUT.

9.1.3.3 Program Execution

Execute programs like ordinary programs. Data input is requested from a standard input when an ACCEPT statement in a program is executed. Enter data as required. When a DISPLAY statement in the program is executed, data is output to the standard output.

Note

Output to the file using the redirection function, or output to the file when you execute the program to the standard output (stdout) in the background. Refer to "9.1.6 Programs Using Files" when modifying the program.

9.1.3.4 Numeric Data Input

Numeric data can be entered by describing numeric data items for receiving data items (external decimal, internal decimal, and binary data items) in the ACCEPT statement. Numeric data for input can be up to 21 digits from the beginning of the input line for hardware devices including return and line feed keys. The format of numeric data input is:

```
$  [sign-character] numeric-digit
```

or

```
$  numeric-digit [sign-character]
```

Numeric digit

Numeric characters from 0 to 9 and decimal points (.) can be specified. Input data is scaled by decimal position corresponding to data item formats specified in the ACCEPT statement and stored.

Sign character

The sign characters "+" or "-" can be specified.

9.1.4 Programs Using Interstage Business Application Server general log

This section explains how to code and execute programs that use the Interstage Business Application Server general log. It provides notes on compilation and link. For an overview of Interstage Business Application Server and details on how to install and use Interstage Business Application Server, see the Interstage Business Application Server manual.

9.1.4.1 Program Specifications

This section explains the descriptions in each COBOL division in the program that uses the Interstage Business Application Server general log.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
   CONSOLE IS mnemonic-name.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  data-name  ... .
 PROCEDURE DIVISION.
     DISPLAY  data-name UPON mnemonic-name.
     DISPLAY  "nonnumeric-literal" UPON mnemonic-name.
     EXIT PROGRAM.
 END PROGRAM  program-name.
```

Environment division (ENVIRONMENT DIVISION)

The environment division requires no special descriptions.

Data division (DATA DIVISION)

The data division requires no special descriptions.

Procedure division (PROCEDURE DIVISION)

To output data to the Interstage Business Application Server general log, use the DISPLAY statement with the mnemonic name associated with the SYSOUT, SYSERR, or CONSOLE function name specified in the UPON phrase. If a data name is specified in the DISPLAY statement, the data stored in the data name is output. If a nonnumeric literal is specified in the DISPLAY statement, the specified character string is output. The general log is opened at the start of program execution, and the DISPLAY statement in the procedure only outputs data to the log. The log is closed at the end of program execution.

9.1.4.2 Program Compilation and linkage

No special compile and link options are required.

9.1.4.3 Program Execution

To execute a program that uses the Interstage Business Application Server general log as the output destination, environment variables must be specified as follows depending on the UPON phrase of the DISPLAY statement:

If the UPON phrase is not specified or the output destination of the SYSOUT function name is used for the general log:

```
$ CBR_COMPOSER_SYSOUT=management-name-defined-by-log-definition file; export CBR_COMPOSER_SYSOUT
```

If the output destination of the SYSERR function name is used for the general log:

```
$ CBR_COMPOSER_SYSERR=management-name-defined-by-log-definition file; export CBR_COMPOSER_SYSERR
```

If the output destination of the CONSOLE function name is used for the general log:

```
$ CBR_COMPOSER_CONSOLE=management-name-defined-by-log-definition file; export CBR_COMPOSER_CONSOLE
```

Note

The general log output level of the DISPLAY statement is 5.

9.1.5 Programs Using System Standard Error Output

This section explains how to write, compile, link, and execute programs using a system standard error output (stderr).

9.1.5.1 Program Specifications

This section explains the program descriptions with a system standard error output (stderr) 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 ... .
PROCEDURE DIVISION.
    DISPLAY  data-name  UPON  mnemonic-name.
    DISPLAY  "nonnumeric-literal"  UPON  mnemonic-name.
    EXIT PROGRAM.
END PROGRAM  program-name.
```

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, associate a mnemonic-name with the function-name SYSERR.

DATA DIVISION

In the DATA DIVISION, define data items to set output data.

PROCEDURE DIVISION

To send data to a standard error output, 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.

9.1.5.2 Program Compilation and Linkage

No specific compiler and linker options are required.

9.1.5.3 Program Execution

Execute programs as ordinary programs. When a DISPLAY statement in the program is executed, data is output to the standard error output. To output data to a file, specify the CBR_MESSOUTFILE environment variable. Refer to "4.3.2 Outputting Error Messages to Files".

9.1.6 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.

9.1.6.1 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 ... .
PROCEDURE DIVISION.
    ACCEPT   data-name.
    DISPLAY  data-name.
    EXIT PROGRAM.
END PROGRAM  program-name.
```

ENVIRONMENT DIVISION

No specifications are required.

DATA DIVISION

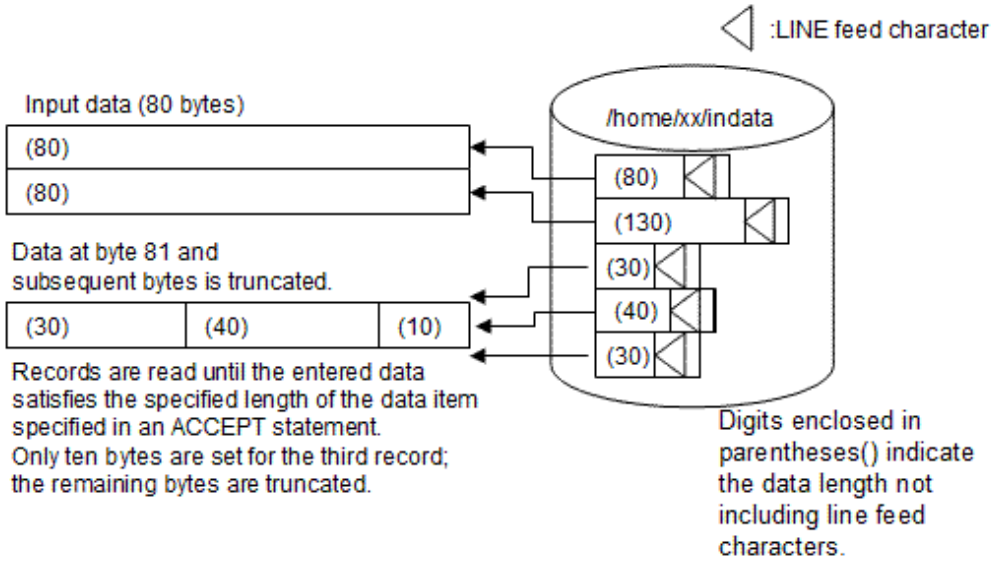
In the DATA DIVISION, define data items to store input data and set output data.

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 input or output only. The input and output files are closed upon termination of program execution.

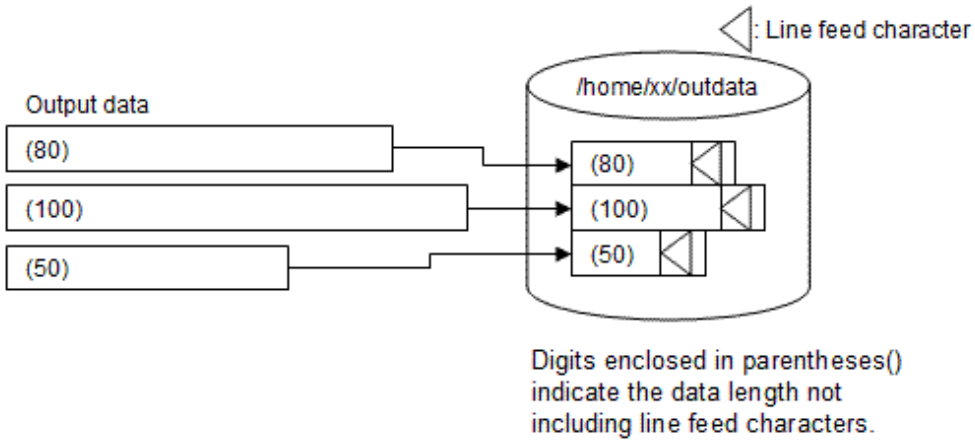
Inputting Data

Use an ACCEPT statement to input data from a file. Data preceding the line feed character is handled as one record. Input data is read by record and 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.



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 output data plus the line feed character is the length of data to be output.



9.1.6.2 Program Compilation and Linkage

To enter data from a file with ACCEPT, specify compiler option SSIN. For more information, see "A.2.44 SSIN (ACCEPT statement data input destination)". To send data to a file with DISPLAY, specify compiler option SSOUT. For more information, see "A.2.45 SSOUT (DISPLAY statement data output destination)".

Example

```
$  cobol -o PROG1 -WC,"SSIN(IN),SSOUT(OUT)" -M PROG1.cob
```

Note

- When SYSIN is specified for the compiler option SSIN, the input destination is the standard input (stdin). The SYSIN environment variable setting is ignored.
- When SYSOUT is specified for the compiler option SSOUT, the output destination is the standard output (stdout). The SYSOUT environment variable setting is ignored.
- When the alias associated with function-name CONSOLE is specified for the description of FROM phrase or UPON phrase, the input/output destination is the standard input (stdin) / output (stdout) regardless of the specification of the compiler option (SSIN/SSOUT).

9.1.6.3 Program Execution

Specify the names of files used for input-output in the environment variable name specified for compiler options SSIN and SSOUT before executing programs.

Example

```
$  IN=/home/xx/indata ; export IN
$  OUT=/home/xx/outdata ; export OUT
```

Note

- The input file is opened in input mode and used in shared mode. Records are not 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 overwritten and previous data is deleted.
- Line feed characters are not handled as data.

- You cannot 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).
- Virtual devices (/dev/null or similar) cannot be used at the I-O destination.
- Refer to COBOL file system in "Table 6.9 Functional differences between file systems", for the maximum file size.

9.1.6.4 Expanded file output functions of the DISPLAY statement

The following expanded functions can be used in file output with the DISPLAY statement:

- Adding data to a file
- Dummy file

This section explains these expanded functions and how to use them.

Adding data to a file

Data can be added to an existing file in output with the DISPLAY statement. The following explains how to use this function.

Usage

In the environment variable specified for the SSOUT compile option, specify ",MOD" after the name of the data output file.

Example

```
$ OUT=/home/xx/outdata,MOD ; export OUT
```

Note

If MOD is specified with an existing file, data is added to the file. If the file does not exist, a file is created.

Dummy file

When the output file is a dummy file, the output file is not generated even though the DISPLAY statement is executed. Refer to "6.8.6 Dummy files" for details.

The dummy file is not generated though the execution of DISPLAY statement succeeds when the output file is made a dummy file.

Usage

In the environment variable specified for the SSOUT compile option, specify ",DUMMY" after the name of the data output file. The file name is optional and can be omitted.

Example

```
$ OUT=/home/xx/outdata,DUMMY ; export OUT
  or
$ OUT=,DUMMY ; export OUT
```

Note

- A comma (,) must precede "DUMMY". If there is no preceding comma, "DUMMY" is treated as a file name. An error doesn't occur.
- If a file name is specified with ",DUMMY", the file name is ignored. Even if the specified file exists, no operation is done to the file.

COMMON Notes

- If the file name contains a comma (,), it must be enclosed in double quotation marks (").
- MOD and DUMMY can be specified at the same time and it does not matter which is specified first. If they are both specified at the same time, the dummy file function is assumed to be specified and the MOD specification is ignored.
- If a character string other than MOD or DUMMY is specified after a comma (,), an error occurs during the initial execution of the DISPLAY statement.

9.1.6.5 Expanded file input functions of the ACCEPT statement

The following expanded functions can be used in file input with the ACCEPT statement:

- Dummy file
- File opening with multithread

This section explains these expanded functions and how to use them.

Dummy file

When the input file is a dummy file, the ACCEPT statement is executed without actually creating an input file.

The execution of ACCEPT statement succeeds even if the input file doesn't exist when the input file is made a dummy file. Refer to "6.8.6 Dummy files" for details.

The value of the data items specified for the ACCEPT statement is not updated.

Usage

In the environment variable specified for the SSIN compile option, specify ",DUMMY" after the name of the data output file. The file name is optional and can be omitted.

Example

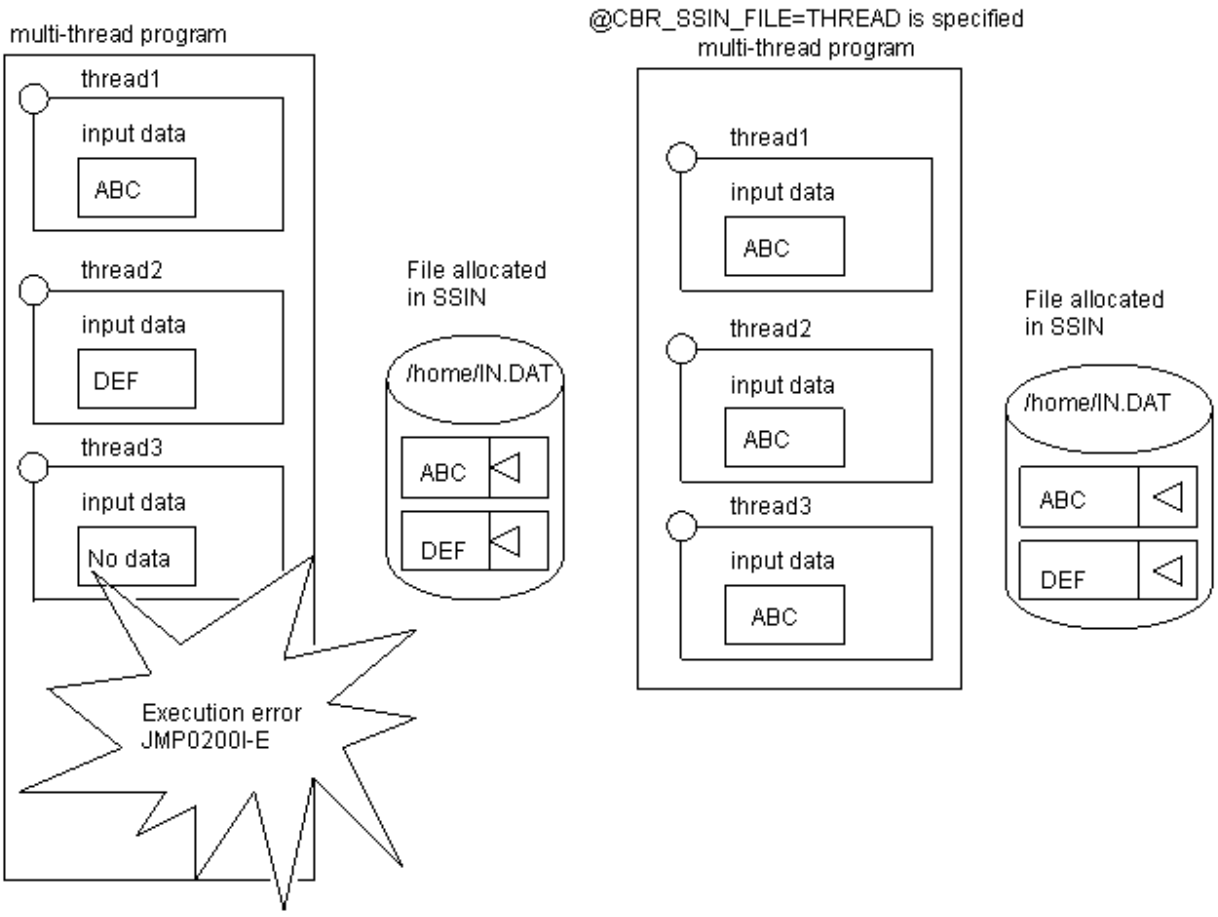
```
$ IN=/home/xx/indata,DUMMY ; export IN
  or
$ IN=,DUMMY ; export IN
```

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 a comma (,), an error occurs during 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.



Usage

Specify "THREAD" for environment variable CBR_SSIN_FILE.

Example

```
$ CBR_SSIN_FILE=THREAD; export CBR_SSIN_FILE
```

9.1.7 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.

9.1.7.1 Programs Specifications

This section explains program descriptions for entering the current date and time based on the system clock by using the ACCEPT statement.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  t-date             PIC 9(6).
  01  t-day              PIC 9(5).
  01  t-day-of-week      PIC 9(1).
  01  t-time             PIC 9(8).
PROCEDURE DIVISION.
    ACCEPT   t-date             FROM DATE.
    ACCEPT   t-day              FROM DAY.
```

```
    ACCEPT   t-day-of-week      FROM DAY-OF-WEEK.
    ACCEPT   t-time             FROM TIME.
    EXIT PROGRAM.
END PROGRAM  program-name.
```

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 phrase.

9.1.7.2 Program Compilation and Linkage

No specific compiler and linker options are required.

9.1.7.3 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

Coding of FROM Phrase	Contents Set for the Data-Name
FROM DATE	941223
FROM DAY	94357
FROM DAY-OF-WEEK	5
FROM TIME	14154500

9.1.8 Entering any date

This section explains how to code and execute a program that uses ACCEPT/DISPLAY to input any date, and it provides notes on compilation and linkage.

9.1.8.1 Programs Specification

This section explains the descriptions in each COBOL division in the program that uses ACCEPT/DISPLAY to input any date.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. program-name.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  year-month-day    PIC 9(6).
 PROCEDURE DIVISION.
     ACCEPT   year-month-day    FROM DATE.
     EXIT PROGRAM.
 END PROGRAM  program-name.
```

Environment division (ENVIRONMENT DIVISION)

The environment division requires no special descriptions.

Data division (DATA DIVISION)

The data division defines data items for storing input data.

Procedure division (PROCEDURE DIVISION)

To enter any date, use the ACCEPT statement with DATE (year.month.day) specified in the FROM phrase.

9.1.8.2 Compiling and linking a program

No special compile and link options are required.

9.1.8.3 Executing a program

To execute a program for input of any date, the following environment variable must be specified:

```
$ CBR_JOBDATE=year.month.day; export CBR_JOBDATE
```

Specify year.month.day as follows:

- year : (00-99) or (1900-2099)
- month : (01-12)
- day : (01-31)

For a year in the 1900s, specify the four digits or last two digits of the year in "year." For a year in the 2000s, specify the four digits of the year in "year."

 Example

Specifying October 1, 1990 as the date

$ CBR_JOBDATE=90.10.01

FROM clause format	Setting in data name
FROM DATE	901001

Specifying October 1, 2004 as the date

$ CBR_JOBDATE=2004.10.01

FROM clause format	Setting in data name
FROM DATE	041001

9.1.9 Programs Using Syslog

This section explains how to write, compile, link, and execute programs that use Syslog.

9.1.9.1 Programs Specification

The code required in each COBOL division in order to write to Syslog 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 (ENVIRONMENT DIVISION)

Associate the mnemonic-name with SYSOUT, SYSERR, or CONSOLE.

Data division (DATA DIVISION)

Define the data to be written to Syslog.

Procedure division (PROCEDURE DIVISION)

To write to Syslog, use a DISPLAY statement where the mnemonic-name associated with SYSOUT, SYSERR or CONSOLE is specified in UPON.

9.1.9.2 Compiling and linking a program

No special compile and link options are required.

9.1.9.3 Executing a program

Specific environment variable settings are required at execution time when using UPON in DISPLAY statements to write to Syslog.

For No UPON or SYSOUT

```
CBR_DISPLAY_SYSOUT_OUTPUT=SYSLOG
```

For SYSERR

```
CBR_DISPLAY_SYSERR_OUTPUT=SYSLOG
```

For CONSOLE

```
CBR_DISPLAY_CONSOLE_OUTPUT=SYSLOG
```

Note

- The maximum length of the Syslog contents that can be output at a time depends on the operating system. For NetCOBOL, use a maximum of 1022 bytes.
- 1-byte code type data can be output to Syslog. Multi-byte code type data is not guaranteed.
- Syslog cannot be used simultaneously with the Interstage Business Application Server General Log.
- You cannot output to any other destinations (standard output, file, etc.) while writing to the Syslog.
- When writing to Syslog, the Identity Name is "NetCOBOL Application". The Identity Name can be changed using the following environment variables:

  If using No UPON or SYSOUT

  ```
  CBR_DISPLAY_SYSOUT_SYSLOG_IDENT=identity-name
  ```

  If using SYSERR

  ```
  CBR_DISPLAY_SYSERR_SYSLOG_IDENT=identity-name
  ```

  If using CONSOLE

  ```
  CBR_DISPLAY_CONSOLE_SYSLOG_IDENT=identity-name
  ```

- The Syslog logging level defaults to "Information". This level can be changed using the following environment variables:

  If using No UPON or SYSOUT

  ```
  CBR_DISPLAY_SYSOUT_SYSLOG_LEVEL = { I | W | E }
  ```

  If using SYSERR

  ```
  CBR_DISPLAY_SYSERR_SYSLOG_LEVEL = { I | W | E }
  ```

  If using CONSOLE

  ```
  CBR_DISPLAY_CONSOLE_SYSLOG_LEVEL = { I | W | E }
  ```

  - I : Information
  - W : Warning
  - E : Error

  Output is directed so Syslog according to the settings of the Syslog parameters. Confirm the Syslog settings in syslog.conf if messages are not output per environment variable specifications.

9.2 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.

9.2.1 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, use ACCEPT FROM the ARGUMENT-NUMBER name. To set the argument number for the next argument value to be retrieved, DISPLAY the argument number UPON the ARGUMENT-NUMBER name. To retrieve an argument value, use ACCEPT FROM the ARGUMENT-VALUE name. A character string delimited with blank spaces or double-quotation marks (" ") is counted as one argument.

9.2.2 Program Specifications

This section explains program descriptions for each COBOL division when the command line argument-handling functions are used.

```
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 receive values.

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.
   02  argument-value       PIC X(10)  OCCURS 1 TO 10 TIMES
                  DEPENDING ON  number-of-arguments.
```

PROCEDURE DIVISION

First, find 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 is set in the data name specified in the ACCEPT statement.

To retrieve an argument value, specify the ordinal position of the argument using the DISPLAY statement ([1]) corresponding to the function name ARGUMENT-NUMBER. The position of the argument is relative to 0 beginning with the command name. Subsequent arguments are specified in order after the command, starting from 1. The value of the data item or the numeric literal specifying in the DISPLAY statement becomes the next argument position.

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. If written without ON EXCEPTION in the ACCEPT statement, an error message is output and the next ACCEPT statement is executed.

```
   DISPLAY  5                  UPON mnemonic-name-1.   *> [1]
   ACCEPT   argument-value(5)  FROM mnemonic-name-2    *> [2]
     ON EXCEPTION MOVE  5 TO  error-number             *> [3]
                  GO TO error-processing
   END-ACCEPT.
```

Note

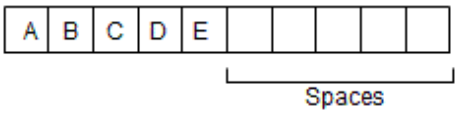
- When a program references an argument value without executing the DISPLAY statement for positioning, the argument position 1 is assumed. Subsequent executions of an ACCEPT statement will increment the value.
- The length of an argument value cannot be obtained.

- 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:

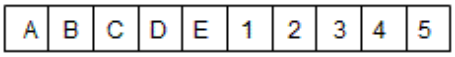
```
*>      :
 01  number-1      PIC 9.
 01  argument-1    PIC X(10).
*>      :
    ACCEPT   number-1 FROM argument-number.      *>[1]
    ACCEPT   argument-1 FROM argument-value.     *>[2]
*>      :
```

If statement (1) is executed when the number of arguments specified in the command is 10, the content 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 padded as follows:



If statement (2) is executed when the value of an argument to be fetched is "ABCDE12345FGHIJ", the content of "argument-1" is truncated as follows:



9.2.3 Program Compilation and Linkage

No specific compiler and linker options are required.

9.2.4 Program Execution

Execute programs as ordinary programs.

Note

- It is recommended to use the command-line retrieval functions with main COBOL programs only.
- If a COBOL program is called from another COBOL program, the values of arguments returned are those of the arguments specified on the command line that activated the application.

9.3 Environment Variable Handling Function

This section explains how to refer to and update the values of environment variables.

9.3.1 Outline

During program execution, the values of environment variables can be referred to and updated using the DISPLAY and ACCEPT statements.

To refer to the value of an environment variable use the following statements:

- A DISPLAY statement using a mnemonic-name corresponding to function-name ENVIRONMENT-NAME.
- An ACCEPT statement using a mnemonic-name corresponding to function-name ENVIRONMENT-VALUE.

To update the value of an environment variable use the following statements:

- A DISPLAY statement using a mnemonic-name corresponding to function-name ENVIRONMENT-NAME.

- A DISPLAY statement using a mnemonic-name corresponding to function-name ENVIRONMENT-VALUE.

9.3.2 Program Specifications

This section explains program descriptions for each COBOL division when using the environment variables handling function.

```
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

```
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
    ENVIRONMENT-NAME  IS   mnemonic-name-environment-variable-name
    ENVIRONMENT-VALUE IS   mnemonic-name-environment-variable-value.
```

DATA DIVISION

Define data items to deliver values.

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 [1] in the code below), where a mnemonic-name corresponding to function-name ENVIRONMENT-NAME is specified. There are two ways to specify the name of the environment variable. One is specifying environment variable names in the DISPLAY statement directly with nonnumeric literal. The second is setting environment variable names for data item and specifying the data name in the DISPLAY statement. Refer to the value of the environment variable with an ACCEPT statement ([2]) that specifies a mnemonic-name that corresponds to the function-name ENVIRONMENT-VALUE. 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-existent environment variable has been specified, an exception condition occurs. In this case, a statement ([3]) specified for ON EXCEPTION is executed. If ON EXCEPTION has not been specified in the ACCEPT statement, an error message is output and the next ACCEPT statement 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. The specification methods of the

environment variable name updated is the same as the environment variable name setting for reference above. Then use a DISPLAY 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. If ON EXCEPTION has not been specified in the DISPLAY statement, an error message is output and the next DISPLAY statement is executed.

```
 DISPLAY "TMP1" UPON  mnemonic-name-environment-variable-name.       *> [1]
 ACCEPT value-of-TMP1 FROM  mnemonic-name-environment-variable-value *> [2]
   ON EXCEPTION                                                      *> [3]
         MOVE   occurrence-of-error TO   ...
 END-ACCEPT.
*>   :
 DISPLAY "TMP2" UPON mnemonic-name-environment-variable-name.        *> [4]
 DISPLAY value-of-TMP2 UPON mnemonic-name-environment-variable-value *> [5]
   ON EXCEPTION                                                      *> [6]
         MOVE   occurrence-of-error TO   ...
 END-DISPLAY.
```

The length of an environment variable cannot be obtained.

9.3.3 Program Compilation and Linkage

No specific compiler and linker options are required.

9.3.4 Program Execution

Execute programs as ordinary programs.

The value of an environment variable changed during program execution is valid only in the process being executed by the program and is not valid for the program after the process ends.

Chapter 10 Using SORT/MERGE Statements (Sort-Merge Function)

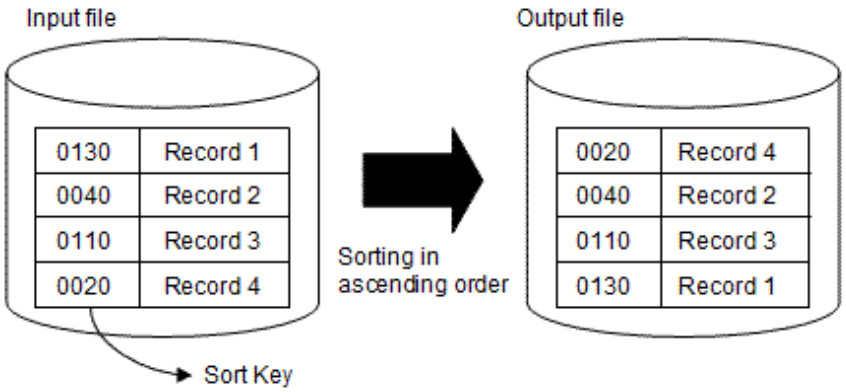
NetCOBOL provides support for sorting and merging records. Sort rearranges unordered records in a file according to a certain sequence, while merge integrates the ordered records of multiple files to a single file. This chapter describes the sort/merge function, looking at the types of sort and merge processing. The following topics are discussed:

10.1 Outline of Sort and Merge Processing

This section outlines sort and merge processing.

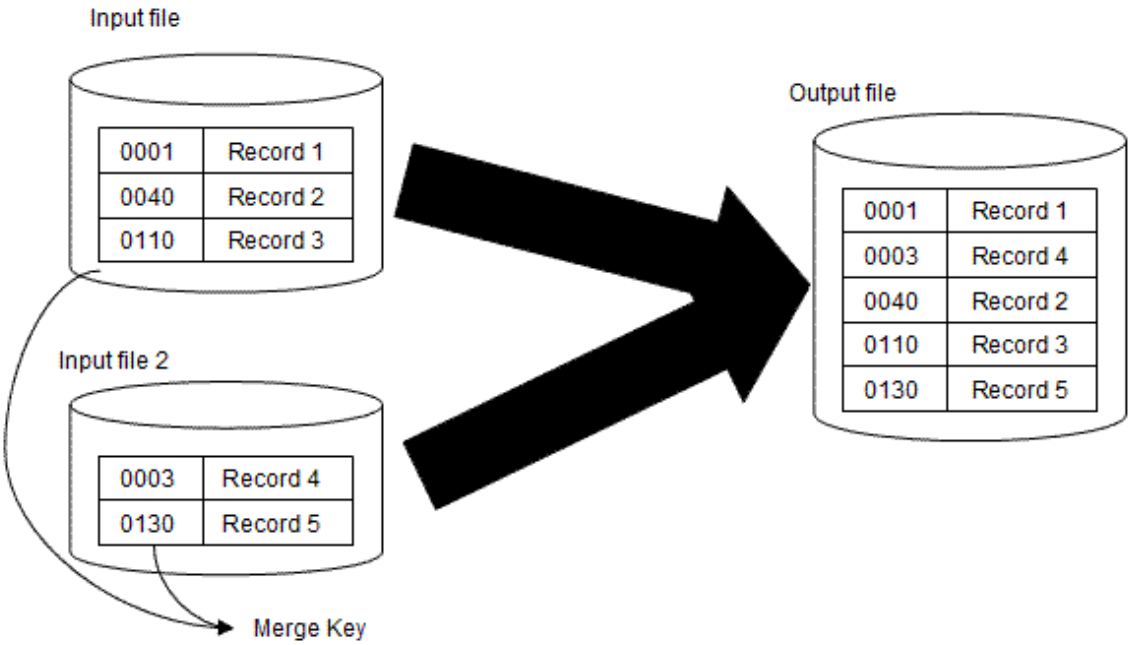
Sort

Sort means that unordered records in a file are rearranged in ascending or descending order using record information as a sort key. The records are rearranged according to the attribute of the program key item.



Merge

Merge means that ordered records in multiple files with records sorted in ascending or descending order are integrated into one file.



10.2 Using Sort

This section explains the types of sort processing, and how to write, compile, link, and execute a program that uses sort processing.

10.2.1 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.

10.2.2 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}.
  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 files must be defined.

Sort-merge file

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.

Note

When performing merge processing within the same program, define only one sort-merge file.

Input file

Define the same way as for ordinary file processing, if required.

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 SORT statement is used for sort processing. The contents of the SORT statement differ depending on what is used for input-output.

- 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.

Following table lists the valid values for special register SORT-STATUS and their meanings.

Table 10.1 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.

The SORT-CORE-SIZE special register enables you to limit the capacity of memory space used by PowerBSORT. This special register is a numeric item implicitly defined in PIC S9(8) COMP-5. The unit of numeric value that is set is byte.

The special register has the same meaning as the value specified in either the SMSIZE() compile option or the smsize execution time option. If SORT-CORE-SIZE, SMSIZE(), and smsize are all specified concurrently, SORT-CORE-SIZE has the highest priority, followed by the smsize execution time option. The SMSIZE() compile option has the lowest priority.

Example

```
Special register     MOVE 102400 TO SORT-CORE-SIZE

(102400 = 100KB)

Compile option   SMSIZE(500K)

Runtime option   smsize300k
```

In this case, the special register SORT-CORE-SIZE value of 100 kilobytes has the highest priority.

10.2.3 Program Compilation and Linkage

Compiler option EQUALS must be specified as 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.

10.2.4 Program Execution

Execute the program that uses sort as follows:

1. Set environment variable BSORT_TMPDIR.

   Sort processing requires a temporary file. A temporary file is created in the directory specified in the BSORT_TMPDIR environment variable.

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

Information

- When PowerBSORT is used, the work file is made in the following directory (shown in priority order):

  1. The directory specified by environment variable BSORT_TMPDIR
  2. The directory specified by BSORT_TMPDIR in the startup file (*)
  3. The directory specified by environment variable TMPDIR
  4. The standard system directory (/tmp)

  * : The default value of PowerBSORT is defined in the startup file. Refer to the PowerBSORT "Online Manual" for details.

- Sort merge processing is usually done by the COBOL runtime system. If PowerBSORT is installed, PowerBSORT is used. PowerBSORT is required when COBOL files of 2G bytes or larger are used in the sort-merge module.

10.3 Using Merge

This section explains the types of merge processing and how to write, compile, link, and execute a program that uses merge processing.

10.3.1 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.

10.3.2 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}.
  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.

When making a merge processing in the same program, define only one file for sort-merge processing.

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.

- For file output, "GIVING output-file-name" must be specified.
- For record output, "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 10.2 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.

The SORT-CORE-SIZE special register enables you to limit the capacity of memory space used by PowerBSORT. This special register is a numeric item implicitly defined in PIC S9(8) COMP-5, with a value specified in bytes.

The register has the same meaning as the value specified in either the SMSIZE() compile option or the smsize execution time option. If SORT-CORE-SIZE, SMSIZE(), and smsize are all specified concurrently, SORT-CORE-SIZE has the highest priority, followed by the smsize execution time option. The SMSIZE() compile option has the lowest priority.

Example

```
Special register     MOVE 102400 TO SORT-CORE-SIZE

(102400=100KB)

Compile option   SMSIZE(500K)

Runtime option   smsize300k
```

In this case, the special register SORT-CORE-SIZE value of 100 kilobytes has the highest priority.

10.3.3 Program Compilation and Linkage

No specific compiler or linkage options are required.

10.3.4 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.

Information

Chapter 11 System Program Functions

This chapter describes the functions that are useful when creating system programs. Included in this chapter are explanations for using pointers, the ADDR and LENG functions, and the PERFORM statement with a NO LIMIT phrase.

11.1 Types of System Program Functions

NetCOBOL has the functions listed below, which are enabled for coding a system program. These functions are called the system program functions in NetCOBOL.

- 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 length, in bytes, of a data item or literal defined by COBOL.

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 is encountered.

11.2 Using Pointers

This section explains how to use pointers.

11.2.1 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).

11.2.2 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, LOCAL-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 allocated 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 implied 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 can be specified in such statements as the MOVE and IF statements just like an ordinary data-name.

11.2.3 Program Compilation and Linkage

No specific compiler and linkage options are required.

11.2.4 Program Execution

No specific environment settings are required.

11.3 Using the ADDR and LENG Functions

This section explains how to use the ADDR and LENG functions.

11.3.1 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.

11.3.2 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 just like an ordinary data-name.

11.3.3 Program Compilation and Linkage

No specific compiler and linkage options are required.

11.3.4 Program Execution

No specific environment settings are required.

11.4 Using the PERFORM Statement with a NO LIMIT phrase

This section explains how to use the PERFORM statement with a NO LIMIT phrase. The topics below explain the following code:

11.4.1 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.

11.4.2 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 from 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, execution continues indefinitely (in an infinite loop).

11.4.3 Program Compilation and Linkage

No specific compiler and linkage options are required.

11.4.4 Program Execution

No specific environment settings are required.

Chapter 12 Introduction to Object-Oriented Programming

This chapter introduces object-oriented programming. It looks at why OO programming has been incorporated in COBOL, considers the goals of object orientated programming, and explains the key concepts involved.

12.1 Overview

Object-oriented (OO) programming introduces various 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.

12.1.1 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.

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.

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.

12.1.2 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.

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.

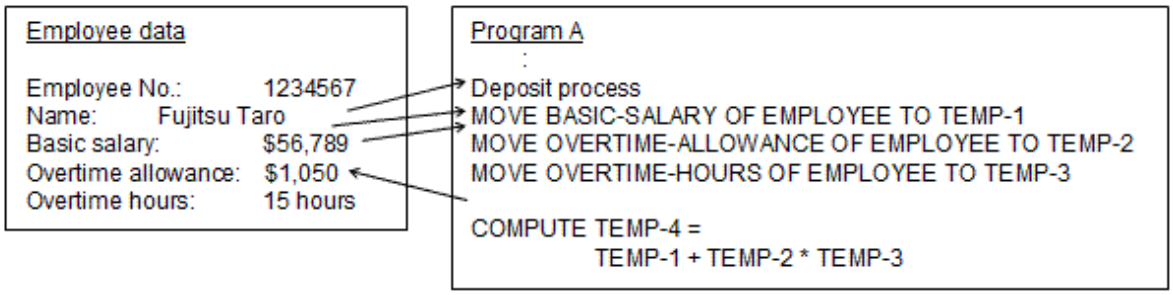
Implementing information hiding in non-OO COBOL has been difficult. For example using PERFORM to execute a logical function requires the function code to know all about the structure of the data. Even when a function has been placed in a separate module, complete record structures are often passed to 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.

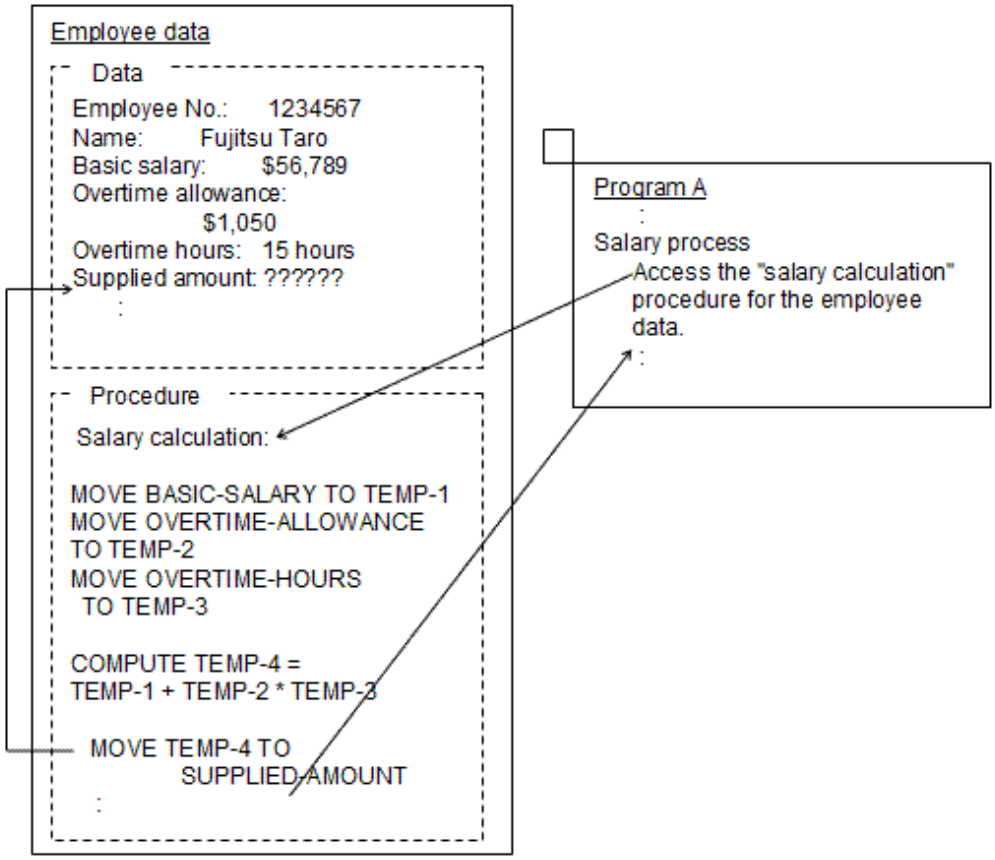
  Figure 12.1 Direct data access



- Example 2 : Information hiding

  Program A only affects the data by invoking the owning procedure (Salary calculation).

  Figure 12.2 Data Hiding



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.

12.1.3 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.

12.1.4 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. NetCOBOL also provides a class browser to help you understand the classes and the inheritance structure.

12.1.5 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.

12.2 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 "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 **abstract classes** 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).

12.3 Concepts of Object-Oriented Programming

12.3.1 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.

12.3.2 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.

12.3.3 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.

12.3.4 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.

12.3.5 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 managing its data.

12.3.6 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.

12.3.7 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.

12.3.8 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.

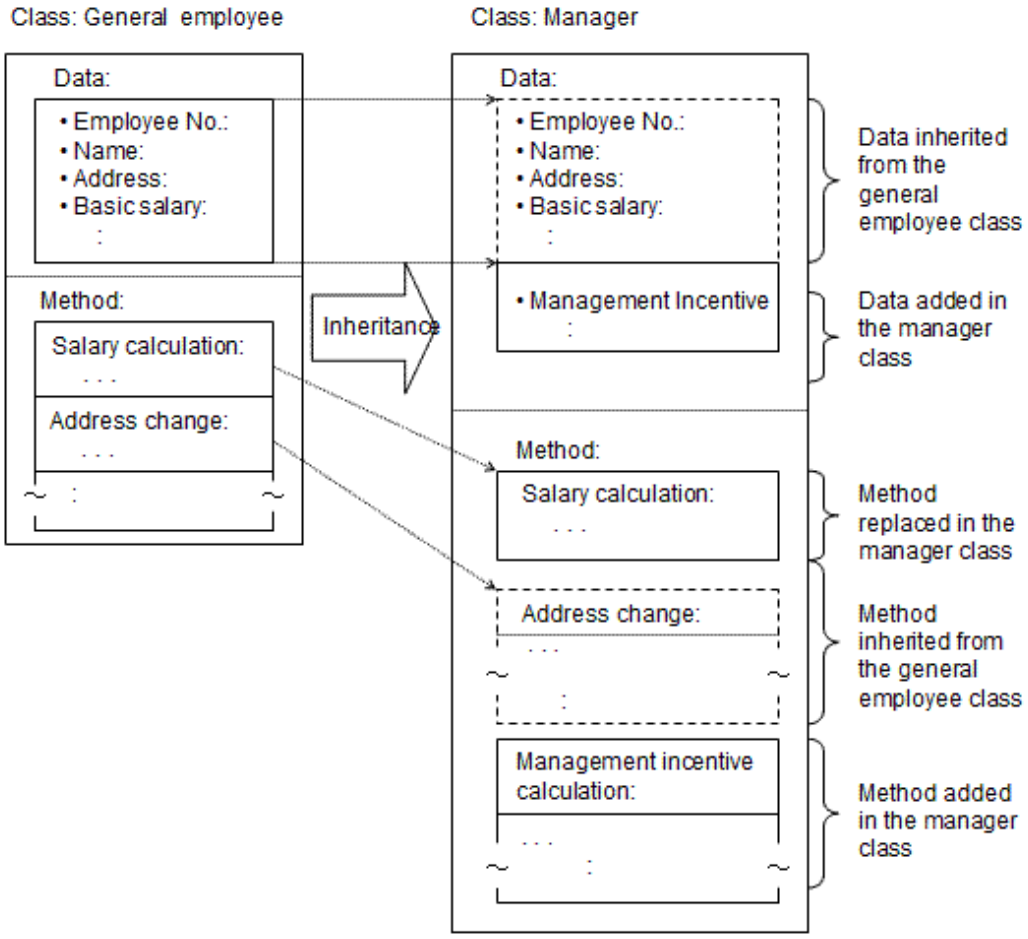
People talk about the data being "encapsulated" in the object.

12.3.9 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 contain 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 12.3 Inheritance example



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.

Single Inheritance

The basic inheritance model has one parent for each child. This is referred to as a single inheritance model.

Multiple Inheritance

OO COBOL supports multiple inheritances 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.

12.3.10 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.

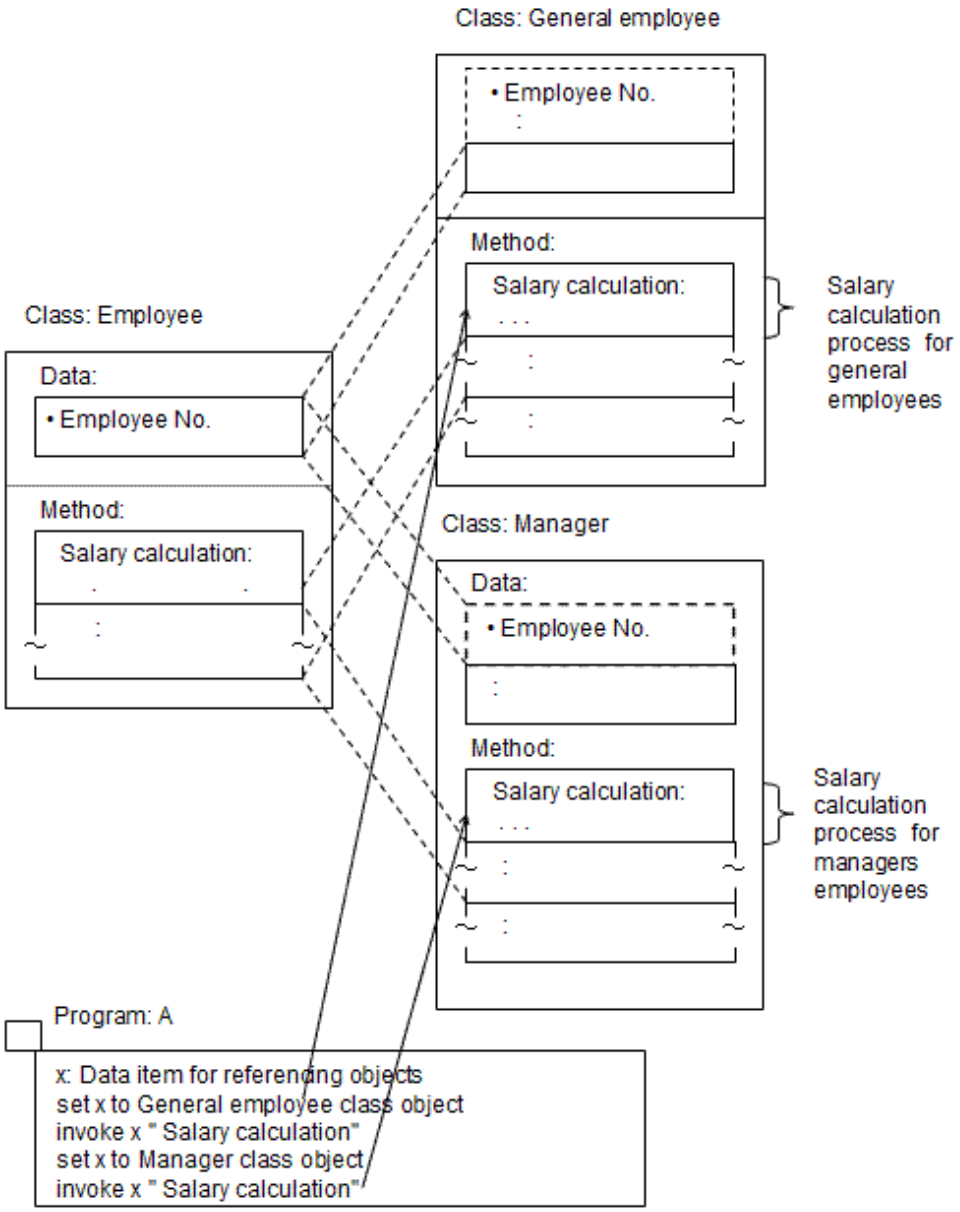
This can bring efficiencies in coding, as one line of code can request a similar function from several different objects, depending on the circumstances when the line is executed.

12.3.11 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, and 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 12.4 Determining correct binding



12.3.12 Conformance

Conformance is the term used in OO programming to describe that an interface or set of interfaces 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.
- Those 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 13 Basic Features of Object-Oriented Programming

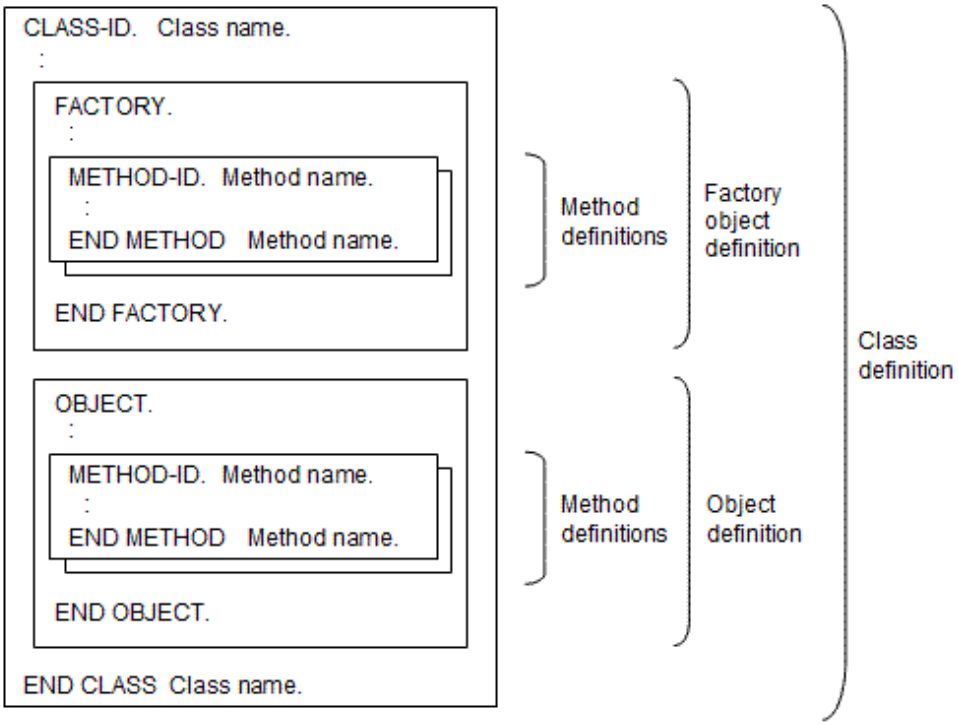
This chapter explains how to implement the Object-Oriented concepts introduced in "Chapter 12 Introduction to Object-Oriented Programming".

13.1 Source Structure

In Object-Oriented programming, the objects and the methods to be operated the object are defined. Object-Oriented COBOL adds the following elements to the standard COBOL program structure:

- Classes definition
- Factory Objects definition
- Objects definition
- Methods definition

These all form part of the class definition which is structured as shown below:

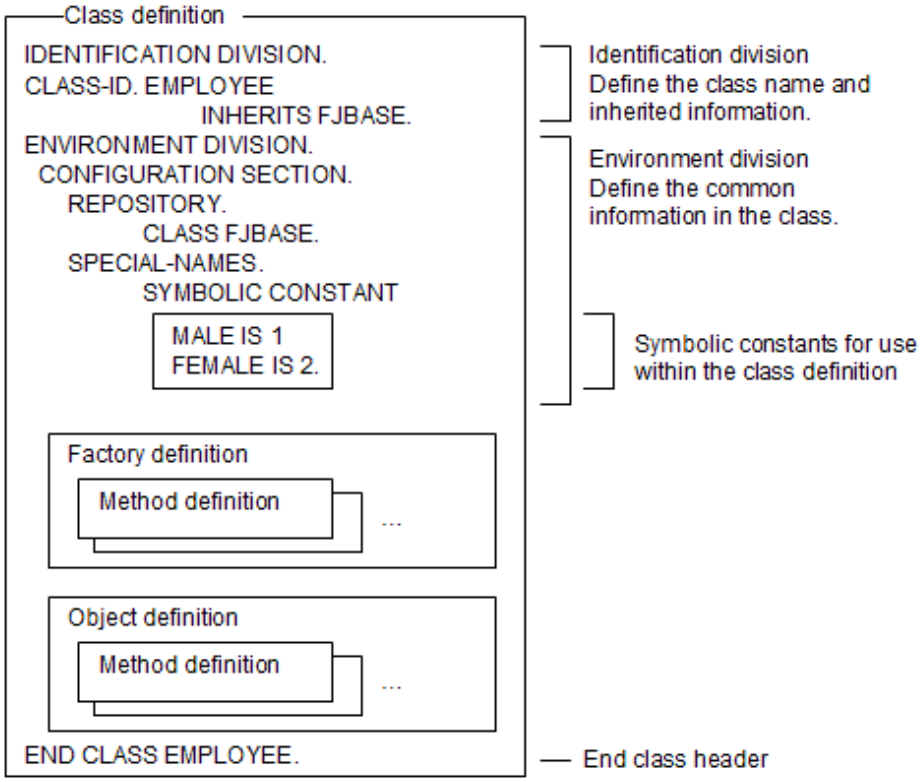


The following explanation is essentially based on the sample (Employee management program) that has been appended to this product. However, to make more easily understood, the data and processes are simplified and the class configuration, methods, data names, and functions that are used in the examples have been changed.

13.1.1 Classes Definition

Class definition is what object definition is based on and encapsulated what data and procedures (methods) to be handled.

Class definitions include factory definitions for managing objects (defining generation and common information and others), defining attributes and formats of objects, and defining object for handling data. In other words, class definition is like the container (frame) that factory definition and object definition are put. Therefore, you can define IDENTIFICATION DIVISION in which class inheritance information is defined and ENVIRONMENT DIVISION in which common information in the class definition is defined. However, you cannot describe data and procedures.



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).

Note

In a program name/class name/method name, some characters cannot be used in the following cases:

- When you specify a program name with a program name paragraph (PROGRAM-ID), CALL statement, and CANCEL statement.
- When you specify a class name and method name with the class name paragraph (CLASS-ID), method name paragraph (METHOD-ID), repository paragraph (REPOSITORY), and INVOKE statement.

The following characters cannot be specified for the above-mentioned cases.

- An underscore (_) as the first character.

Other characters than above-mentioned can be used. However, users should decide whether specified characters act on rules of the linkers.

13.1.2 Factory Definition

A single factory object is created for each class. It provides:

- Data common to objects - "factory data".
- Methods for object management.

The source definition for the factory consists of:

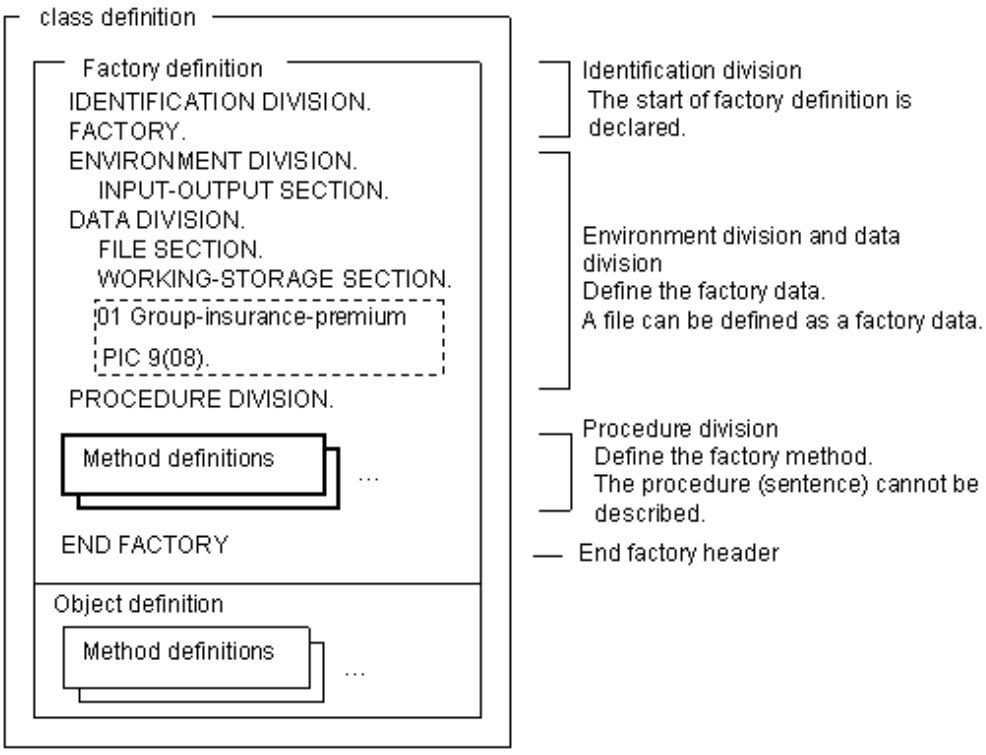
- Identification division containing the FACTORY paragraph.
- Environment and data divisions defining the factory data.
- Procedure division containing the factory methods.
- END FACTORY terminator.

Note

PROCEDURE DIVISION that define only factory methods and procedures (COBOL statements) cannot be described.

Information

Factory definition (from head of factory beginning to the head of factory end) can be omitted for classes that do not need these definitions.



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).

The actual purpose of factory definition is explained below.

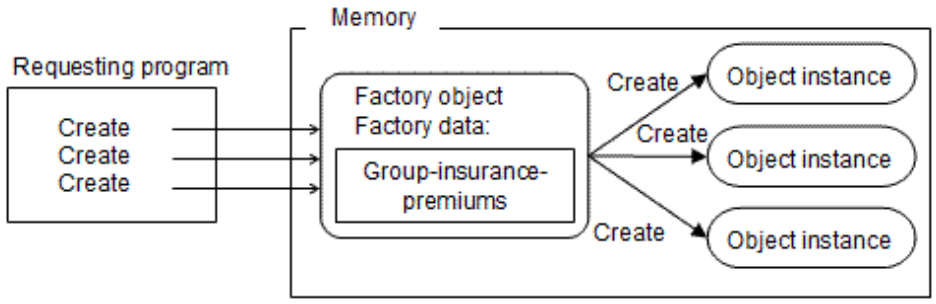
The concept of object instances is explained "12.3.1 Objects". However, the concept of factory objects is included in COBOL.

Factory Objects

Only one factory object exists per class while two or more object instances are generated. Defining this factory object is called factory definition.

Factory objects are generated in a memory on the timing of starting applications and remains in the memory to the end.

For instance, when an object instance is generated, the instruction "Generate" is put out to the class in which the object is defined. However, it is a factory object that receives the instruction. When the factory object receives the instruction, it generates a new object instance.



The above figure explains the operation of "Generation" which is a major function for factory definition. As understood from the figure, factory object is literally "factory" in which object instances are generated.

Users usually need not code "Generation processing" because it is embedded in the class by succeeding to the FJBASE class (see the Note below). What you define then is a method for initializing generated object instances and data to be used in plural object instances that have been generated. For instance, "Group insurance" is defined as factory data as in the example above. This can be the amount that is withdrawn from the salary for all employees when the salary is calculated and can be set and referred by calling the factory method. Maintaining class common information as factory data allows you to correspond to increase and decrease of amount.

Note: Provided natively. For more information, refer to "13.3.2 FJBASE Class".

13.1.3 Objects Definition

The method to handle definition of object data and objects (It is referred to as object method) is defined in the object definition.

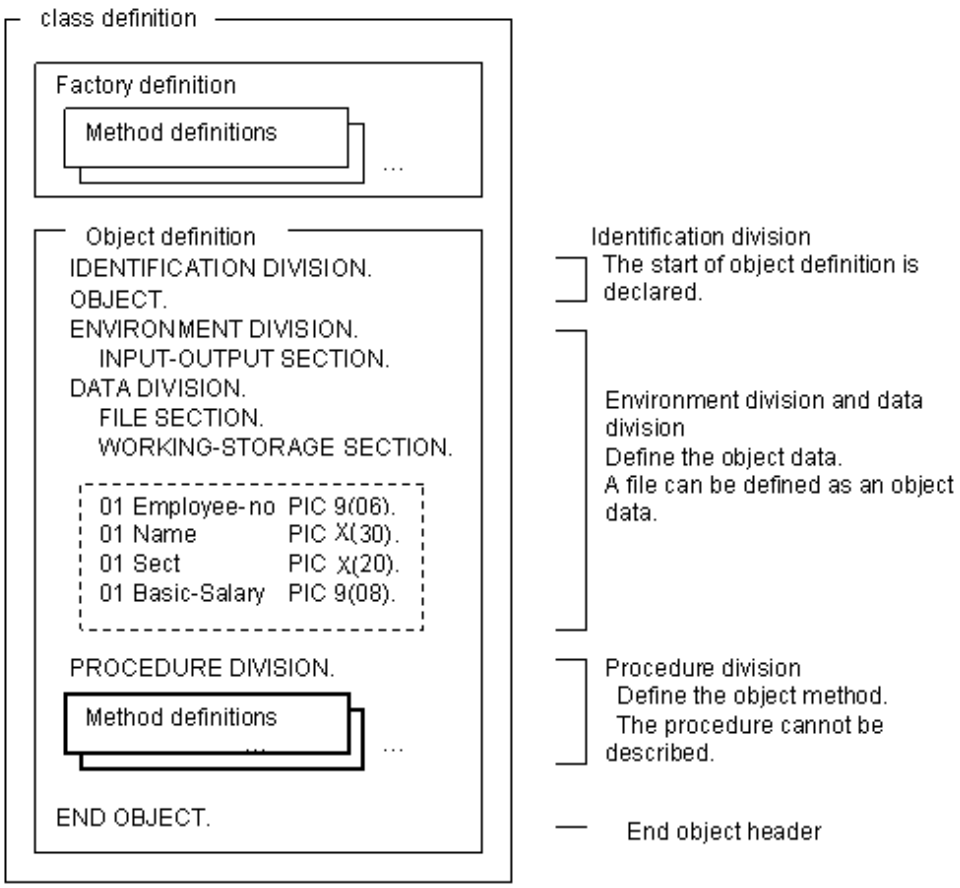
The objects are defined using an identification division with the object header, environment and data divisions for the object data, and a procedure division containing the object methods.

Note

PROCEDURE DIVISION only defines object methods and cannot describe procedures.

Information

Object definition (from head of beginning of the object to the head of end of the object) can be omitted.

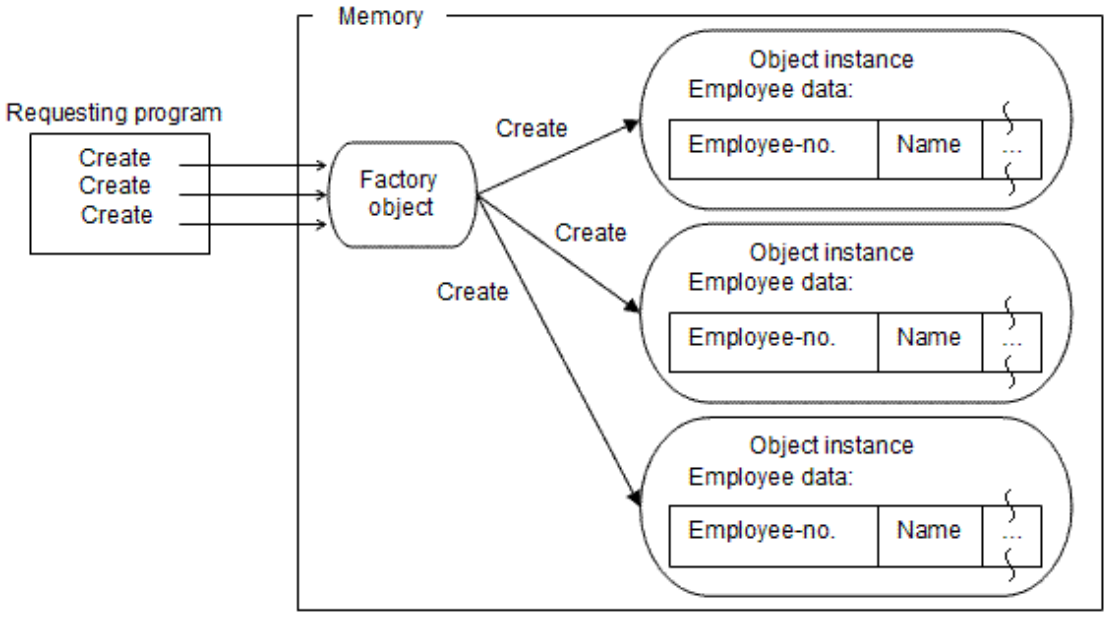


Data defined in the environment and data divisions of the object definition can only be accessed by the object methods (i.e. the data items cannot be referenced outside the bold-line frame in the diagram above).

What must be actually defined for object data and what method to define object methods are explained below.

Define object data in the in the DATA DIVISION (and in ENVIRONMENT DIVISION) and a procedure to handle object data as an object method.

For instance, when data concerning an employee is described, the employee data becomes the object data as in the above figure. In this case, the object instance under execution is as shown in the figure below:

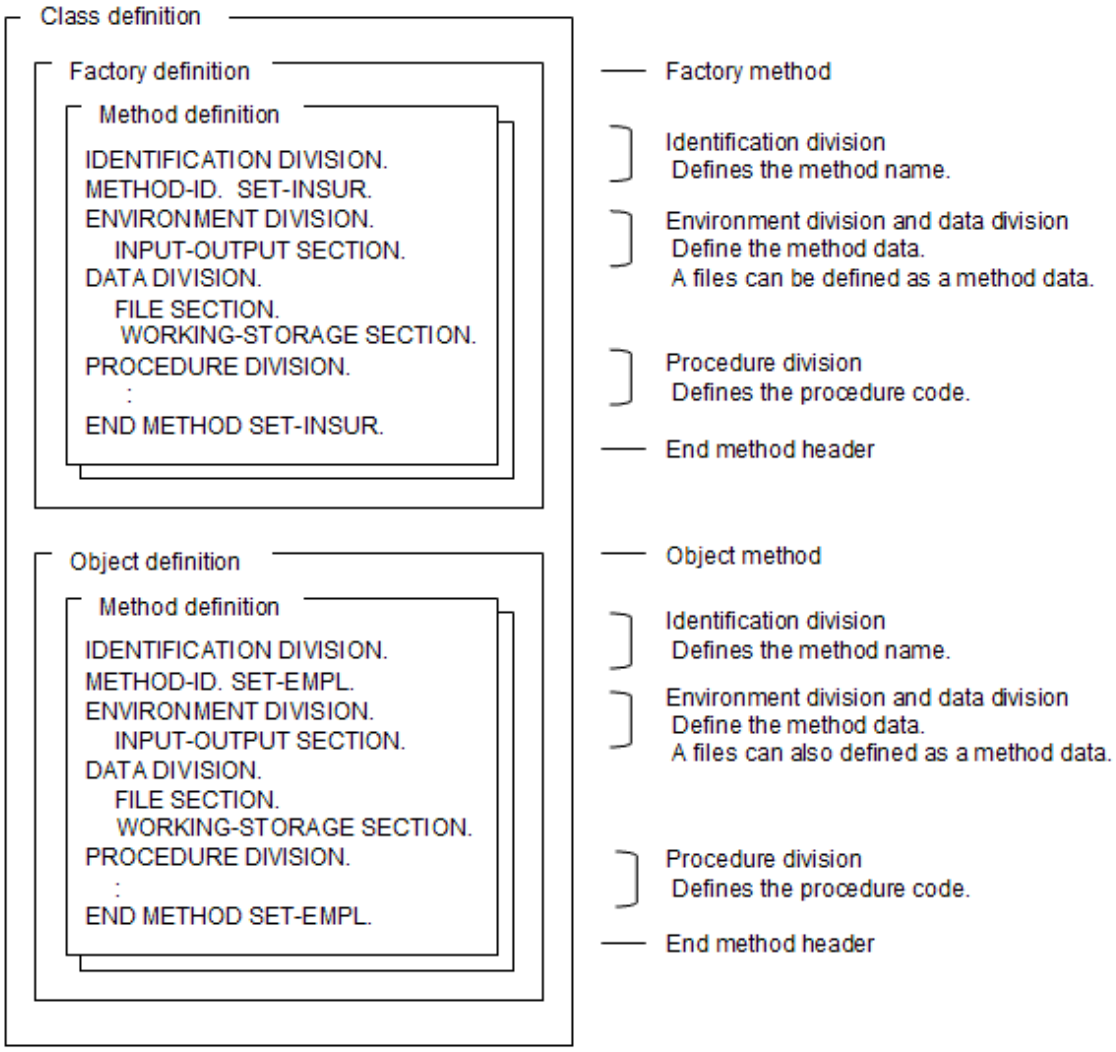


13.1.4 Methods Definition

Defining methods include the factory method for managing object instances and object method for handling object data.

The method definition consists of the IDENTIFICATION DIVISION to define method names, ENVIRONMENT DIVISION and DATA DIVISION to define method data, and PROCEDURE DIVISION to describe procedures. In other words, it can have the same composition as conventional internal programs. Also, factory methods and object methods can be defined respectively as necessary because they have no limitations in the number.

Factory method and object method have the same configurations. Methods defined in factory definition are referred to as factory methods and methods defined in object definition are called object method.



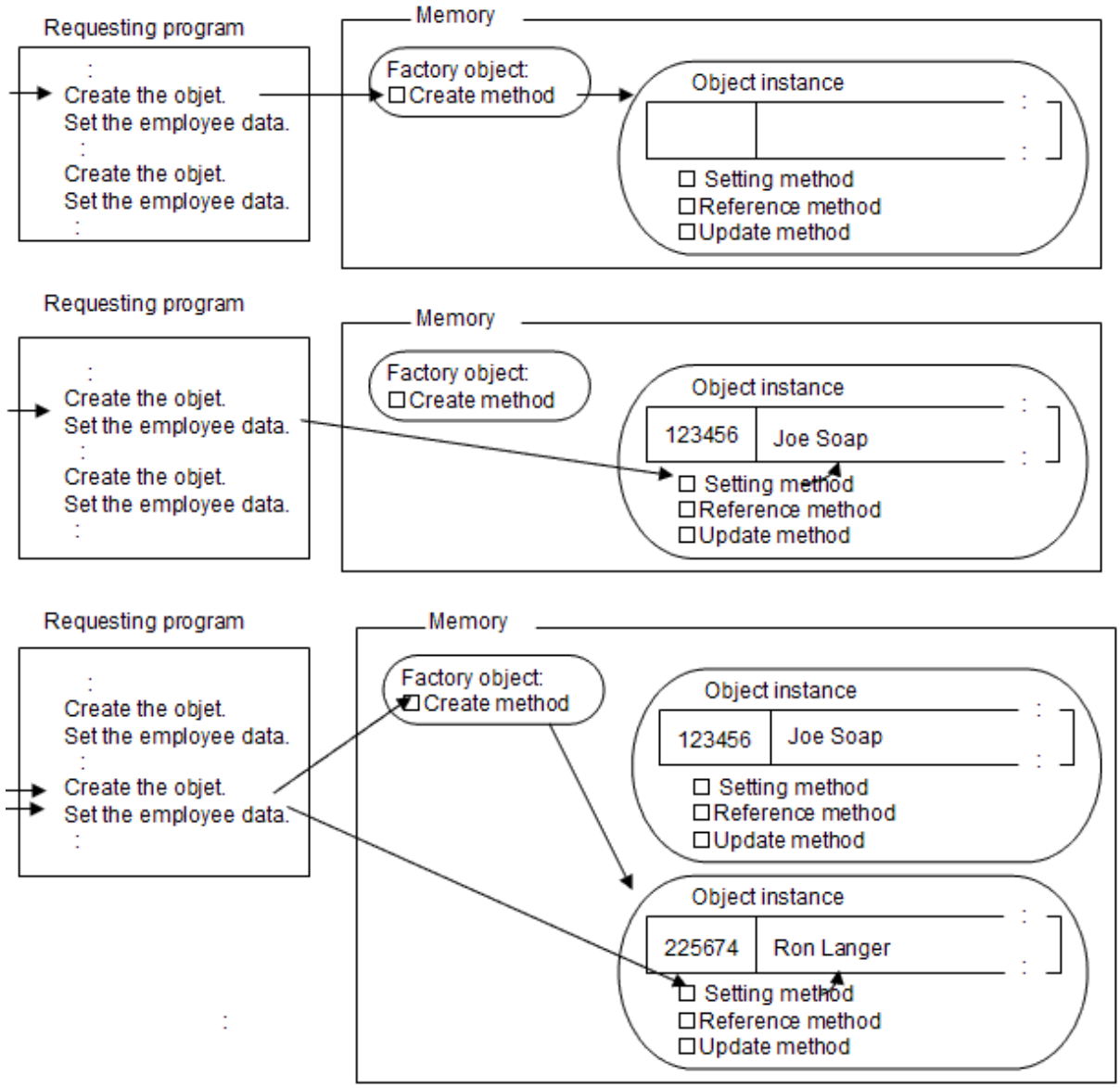
Methods can be factory methods (defined in a factory definition) or object methods (defined in an object definition).

However, data defined in one method can be accessed only in that method.

The following explains the purpose of the methods.

Factory data and object data are concealed from the outside. Data can only be handled (taking out and update it, for example), by the method is the data was defined in for each operation. In other words, factory data can access only via factory methods and object data can only be accessed via object methods.

An image of executing methods can be got more easily if it is considered that a method (procedure) exists in each object instance. Operation of object instances can be that the following image:



13.2 Working with Object Instance

This section explains how to work the object instances.

13.2.1 Invoking Methods

Once an object instance is created, you communicate with that object by invoking its methods. Then, specify which method of which object instance is called. Use the data item called an object reference data item to express which object instance.

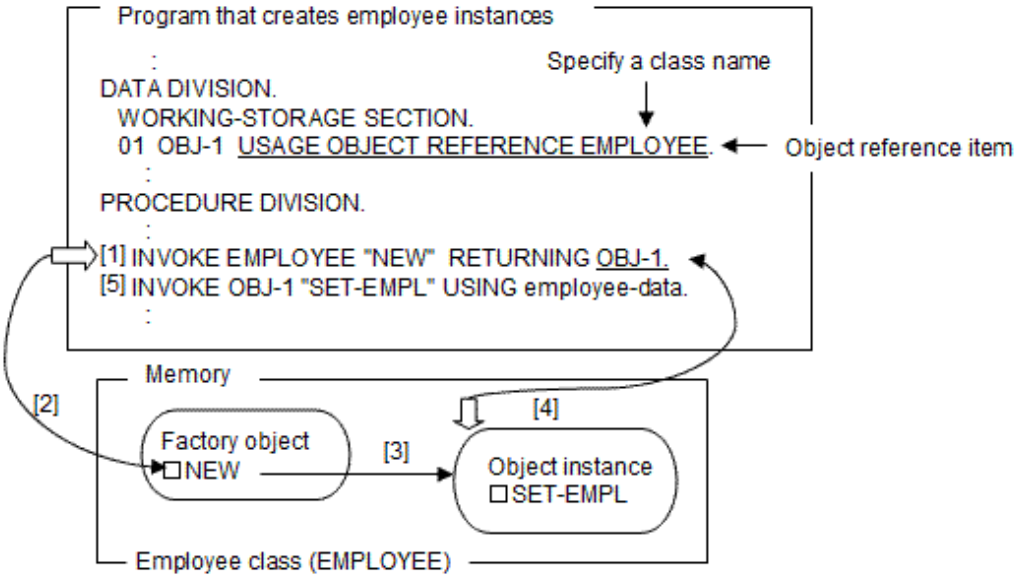
13.2.1.1 Object Reference Data Item

You store object references in data items specified with the USAGE OBJECT REFERENCE clause.

The purpose of this is for storing object references. Therefore, it is mainly used in calling the method (INVOKE statement).

When the method for generating object instances is called, an object reference for the generated object (equivalent to an address) is returned. This object reference data item is subsequently used when this object instance is handled.

Figure 13.1 Creating a new instance of the employee class



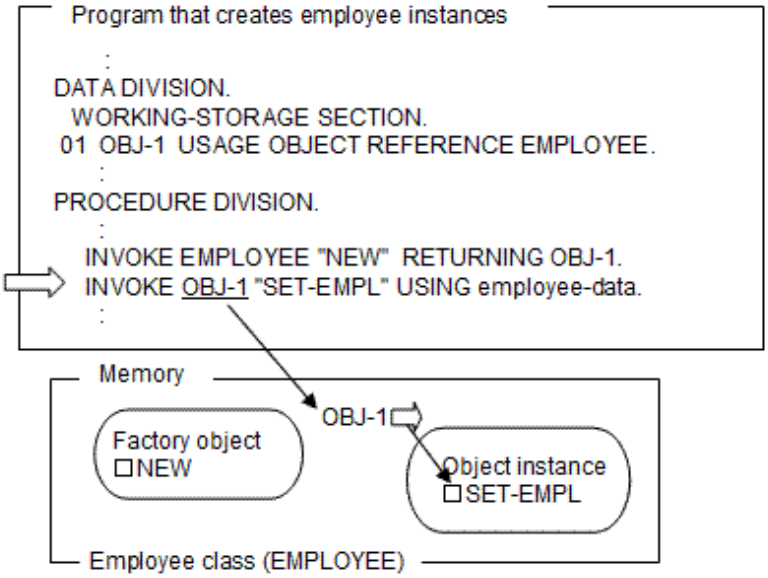
[1] and [2] The creating program invokes the class's NEW method.

[3] The NEW method creates a new instance of the object.

[4] The NEW method returns the object reference in the data item.

[5] The method is invoked using the object reference.

Figure 13.2 Populating the data of an object instance



As shown in "Figure 13.1 Creating a new instance of the employee class" and "Figure 13.2 Populating the data of an object instance", the object reference data item indicating the object instance to be processed should be specified when the generated object instance is operated.

Object reference data items can substitute values for other object reference data items by using the SET statement (they cannot be posted with the MOVE statement). Also, contents can be compared by the IF statement or others. However, note that in this case, object reference data is substituted or compared but object instances are not substituted.

```
  :
DATA DIVISION.
 WORKING-STORAGE SECTION.
```

```
 01  OBJ-1       USAGE OBJECT REFERENCE EMPLOYEE.
 01  OBJ-X       USAGE OBJECT REFERENCE EMPLOYEE.
    :
PROCEDURE DIVISION.
    :
    INVOKE EMPLOYEE "NEW" RETURNING OBJ-1.
    SET OBJ-X TO OBJ-1.
    INVOKE OBJ-X "SET-EMPL" 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 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.

13.2.1.2 INVOKE Statement

For the conventional program, you use the CALL statement for invoking other program. However, when using the method, you must 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 following section explains the INVOKE statement in "Figure 13.1 Creating a new instance of the employee class".

[1] calls NEW method of the EMPLOYEE class to generate object instances. The following example means;

- "Which object" -> Factory object (A factory object is usually expressed by a class name).
- "Which method" -> NEW method.
- "What parameter" -> OBJ-1 (refer to the object) is called as a return value.

[5] calls the SET-EMPL method to set employees' information in the object instances generated in [1] as initial data.

The INVOKE statement for this means as follows:

- "Which object" -> Object instance shown in OBJ-1.
- "Which method" -> SET-EMPL method.
- "What parameter" -> Employee information is called as an input.

Note

When a unique name to identify a method name is specified for the INVOKE statement, the valid maximum character string as a method name is less than 255 characters from the heads of the specified area. Characters after the 256th character are disregarded. Blank spaces at the end of the character string are also ignored.

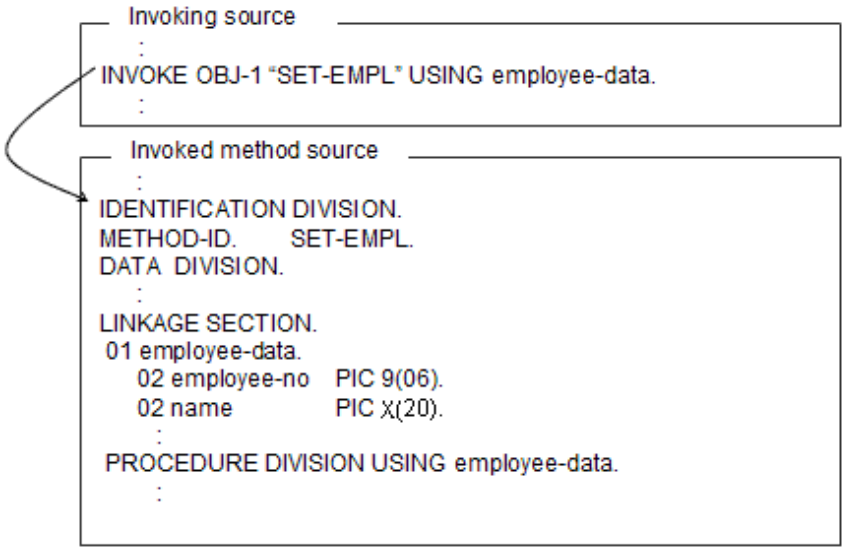
13.2.1.3 Parameter Specification

You can specify parameters for the called method with the INVOKE statement. The CALL statement can specify parameters for the called module.

The parameters are specified with the USING specification and RETURNING phrase.

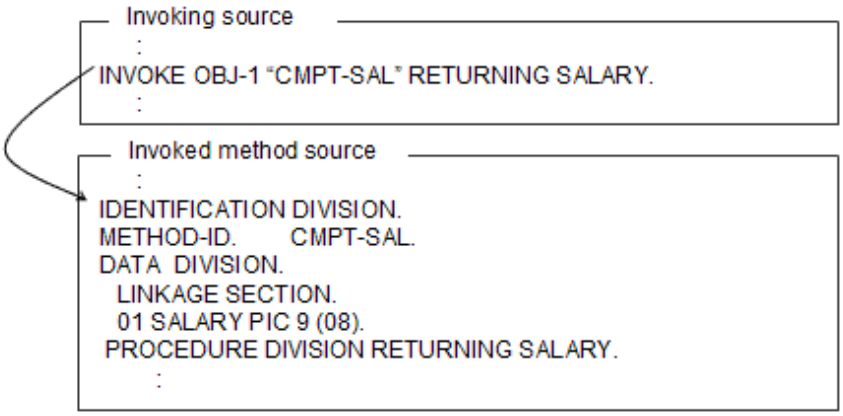
USING phrase

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.

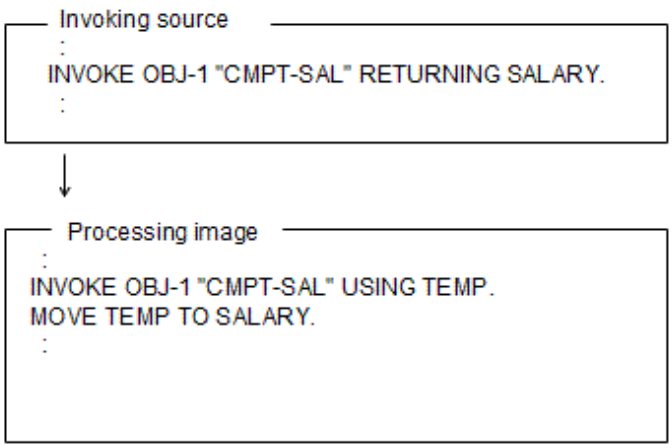


RETURNING Phrase

The RETURNING phrase specifies a single item that can be used to receive a value from the method.



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 phrase generates the following code:



You can use both USING phrase and RETURNING phrase in the same INVOKE statement.

13.2.2 Object Life

"Creating and Referring to Object Instances" explained 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.

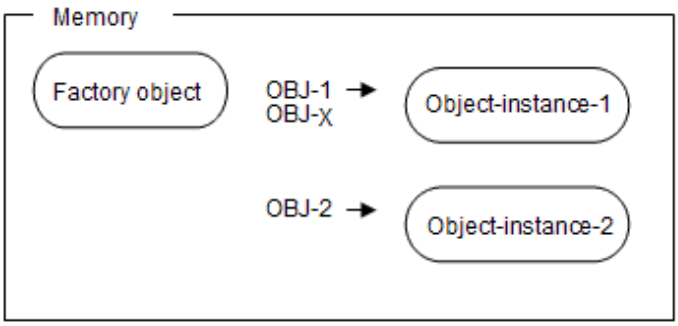
  Note: In this case, _FINALIZE method is not called.

- 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:

[invoking source]

```
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  OBJ-1  USAGE OBJECT REFERENCE EMPLOYEE.
  01  OBJ-2  USAGE OBJECT REFERENCE EMPLOYEE.
  01  OBJ-X  USAGE OBJECT REFERENCE EMPLOYEE.
 PROCEDURE DIVISION.
     INVOKE EMPLOYEE "NEW" RETURNING OBJ-1.
     SET OBJ-X TO OBJ-1.
     INVOKE EMPLOYEE "NEW" RETURNING OBJ-2.
```



When the following procedure is executed in the above condition

```
    SET OBJ-2 TO NULL.          *>[1]
    SET OBJ-1 TO NULL.          *>[2]
    SET OBJ-X TO NULL.          *>[3]
```

Explanation of diagram

- [1] Object-instance-2, referenced only by the object reference data item OBJ-2, is deleted from memory.
- [2] Although obj-1 is set to NULL, object-instance-1 is not deleted because OBJ-X still references it.
- [3] 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).

13.3 Inheritance

The inheritance is one of the key features in Object-Oriented programming. It provides the following benefits:

- Existing components can be reused easily.
- Changes can be accommodated in a flexible manner.

This section explains the inheritance concept and how to use it.

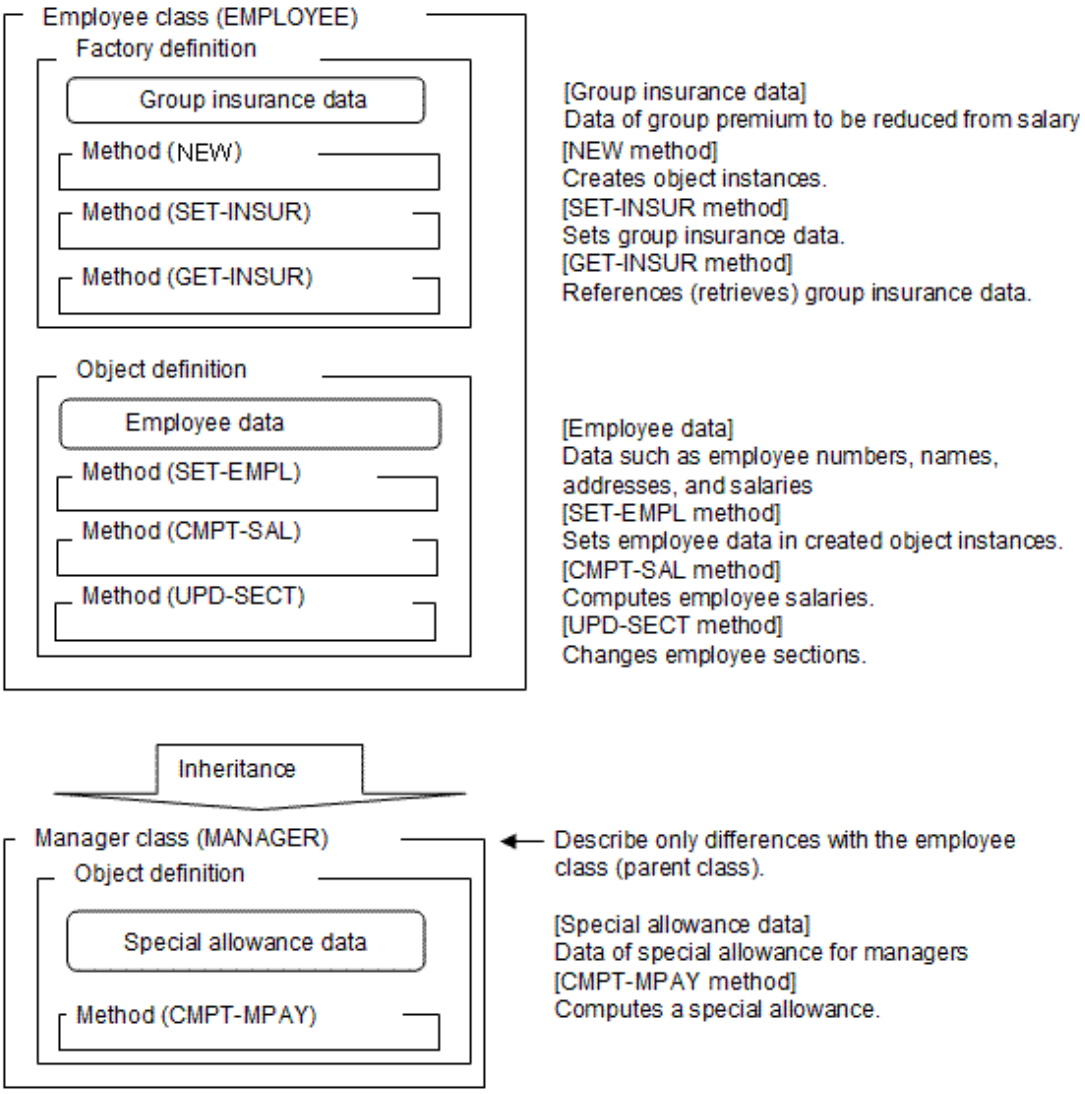
13.3.1 Inheritance Concept and Implementation

Inheritance Concept

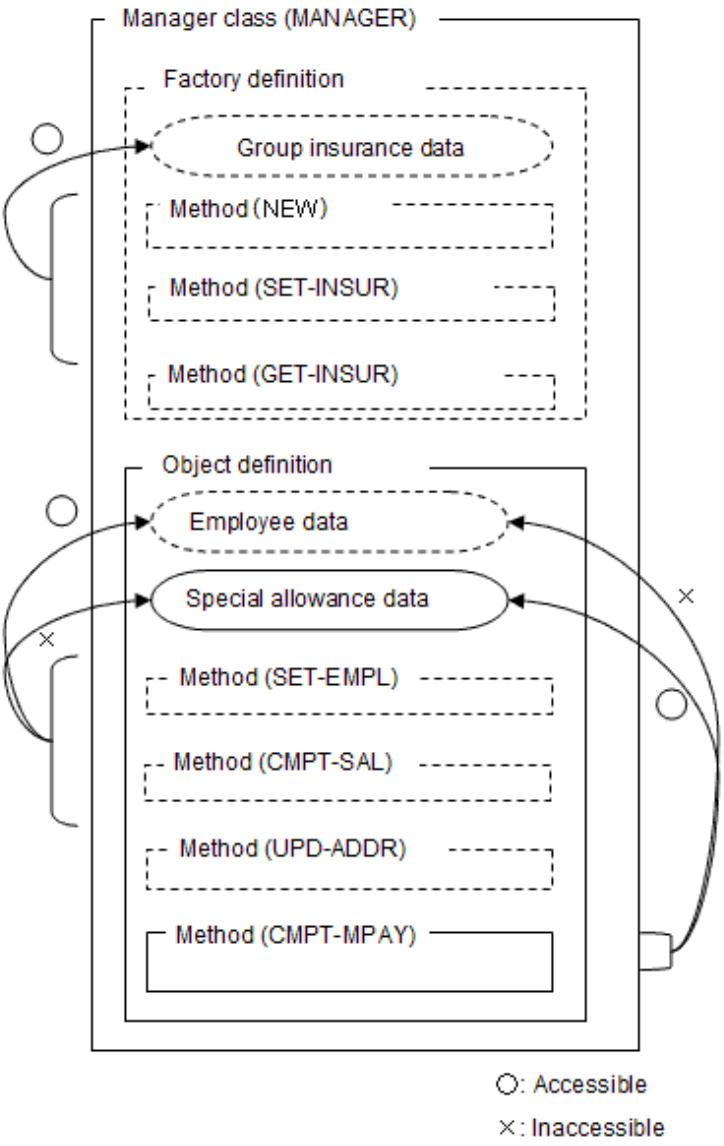
In conventional programming techniques, existing source programs are copied, and new programs are created basis on the copied programs. This is true when a component containing a function similar to that of an existing component (routine) is created.

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.

The following is an example of a method to create a manager class that inherited an employee class:



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; those enclosed in dotted 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). It is recommended to design a hierarchical structure with a moderate depth, because too deep a hierarchy makes it time-consuming to manage resources in that type of structure.

If 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.

How to define inheritance

This section explains how to define inheritance.

Inheritance can be achieved by specifying a parent class name for the INHERITS clause in class name paragraph (CLASS-ID)). Then declare a parent class in the REPOSITORY paragraph in the ENVIRONMENT DIVISION.

```
*>---------------------------
*> CLASS DEFINITION
*>---------------------------
 IDENTIFICATION DIVISION.
 CLASS-ID. MANAGER
    INHERITS EMPLOYEE.    *> Specify the parent class.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
    CLASS EMPLOYEE.       *> Declare the parent class.
*>---------------------------
*> OBJECT DEFINITION
*>---------------------------
 IDENTIFICATION DIVISION.
 OBJECT.
 DATA DIVISION.           *>--+ Object data definition:
 WORKING-STORAGE SECTION. *>  |  Specify the additional methods only.
 01 MPAY PIC 9(08).        *>--+ (Code the difference only)
PROCEDURE DIVISION.
*>---------------------------
*> METHOD DEFINITION
*>---------------------------
IDENTIFICATION DIVISION.  *>--+ Object method definition:
METHOD-ID. CMPT-MAY.      *>  |  Specify the additional methods only.
*> :                      *>--+  (Code the difference only)
END METHOD CMPT-MAY.
END OBJECT.
END CLASS MANAGER.
```

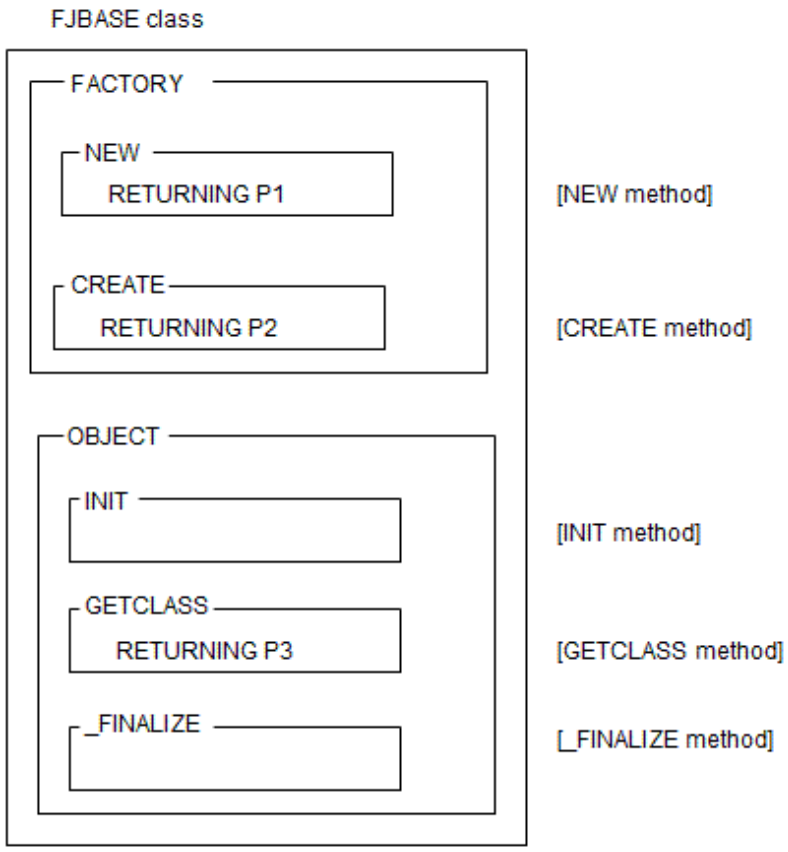
In this way you can easily add functions to existing applications.

13.3.2 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:



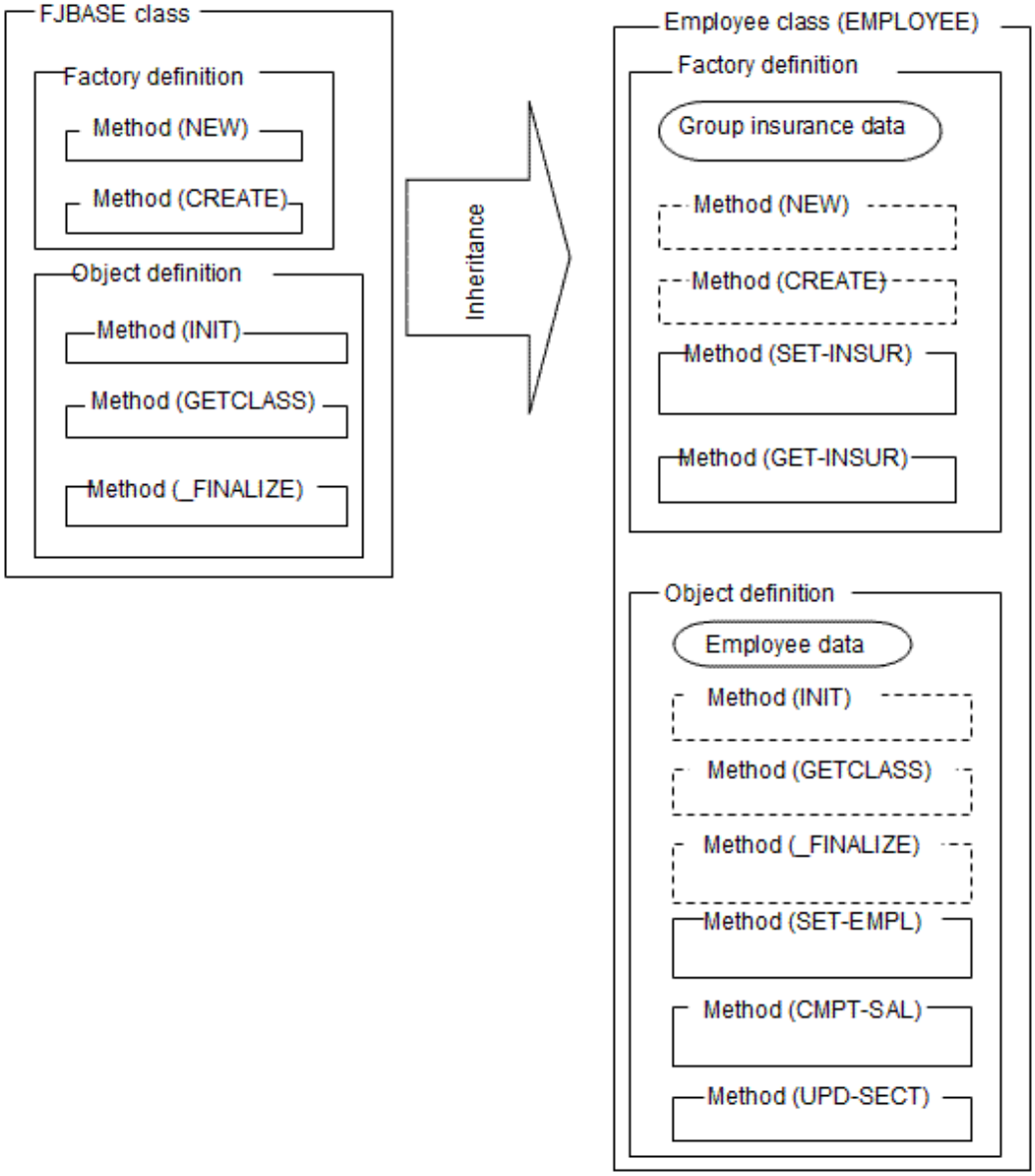
The NEW and GETCLASS methods are the ones most frequently used. Other methods tend to be used in more advanced programming. See "13.7.6 Initialization and Termination Methods" in this chapter for the INIT method and the _FINALIZE method.

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.

For details of the FJBASE class methods, refer to the "NetCOBOL Language Reference".

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.

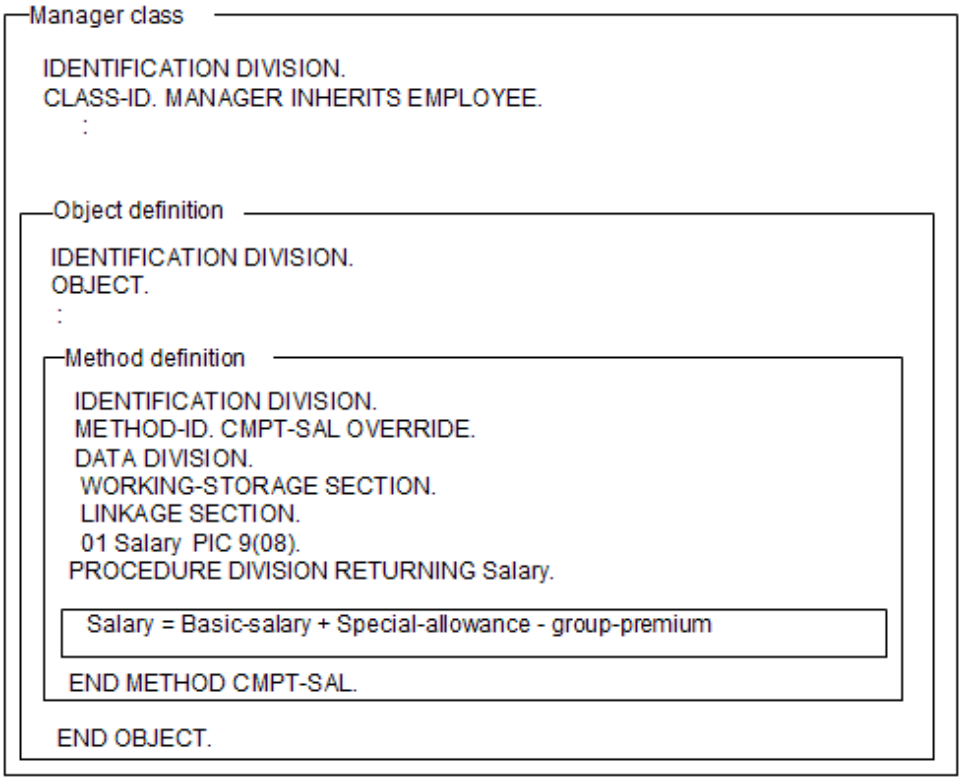


13.3.3 Overriding Methods

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 below:



You can override methods explicitly defined in the parent class or that are implicitly defined in the parent class by inheritance from other classes.

However, the interface for an overriding method must be the same as that for the method being overridden.

13.4 Conformance

Object-Oriented programming includes a concept called conformance. Conformance involves a relationship among classes and you need to consider it when you use (operate) object reference data items.

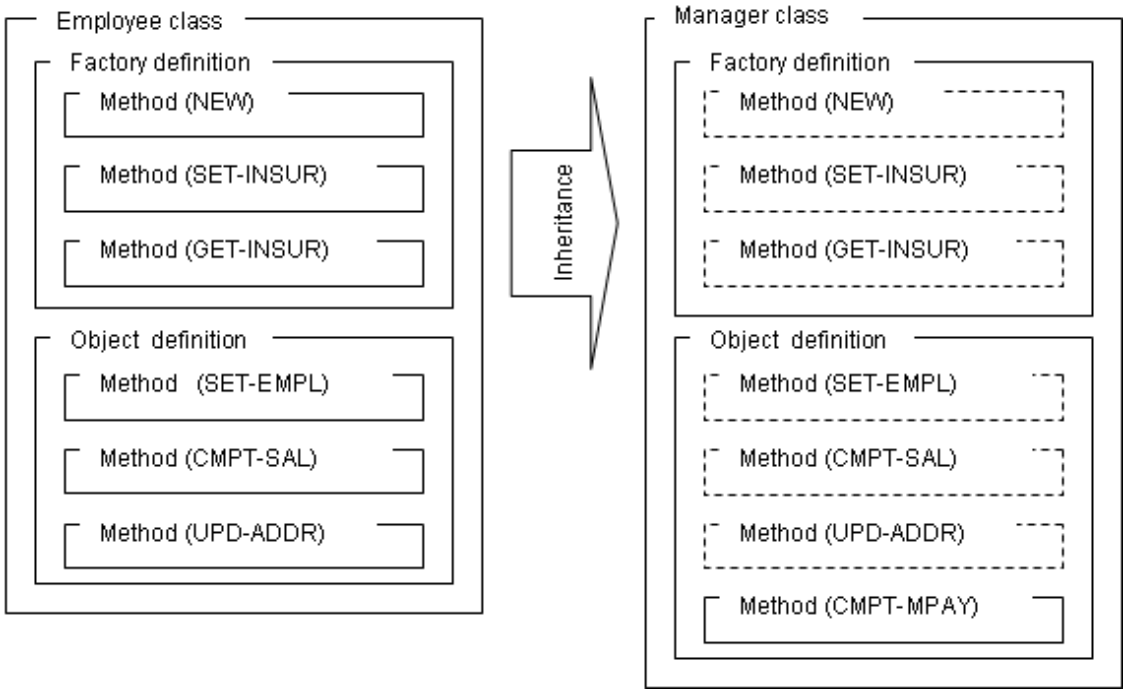
This section explains the concept and rules of conformance using specific examples.

13.4.1 Concept of Conformance

Most Object-Oriented 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:

Conformance refers to a relationship among classes and it can be expressed as "B conforms A" when there is class (B) that contains the interface of a class (A) completely. The interface, in this case, indicates parameters of methods (including implicit definitions) defined in a class and the methods. In other words, conformance relationship is approved in a class in the parent-child relationship by inheritance (the child conforms the parent).

The illustration is as follows:



You can see that the Manager class includes all of the features provided by the Employee class. It can be said that "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. It can be said that "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 "CMPT-SAL" ...
```

will also be valid if the employee-object reference data 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.

```
   :
WORKING-STORAGE SECTION.
01  OBJ-1       USAGE OBJECT REFERENCE EMPLOYEE.
01  OBJ-2       USAGE OBJECT REFERENCE MANAGER.
   :
PROCEDURE DIVISION.
      :
    INVOKE EMPLOYEE "NEW" RETURNING OBJ-1.
    SET OBJ-2 TO OBJ-1.                          ... [1]
      :
    INVOKE MANAGER "NEW" RETURNING OBJ-2.
    SET OBJ-1 TO OBJ-2.                          ... [2]
      :
```

Explanation of the diagram

In the case of [1], a compile error occurs because the conformance relationship from the employee class to the manager class is not approved.

In the case of [2], the object reference is posted without any trouble because the conformance relationship from the employee class to the manager class is approved.

Checks are executed for interfaces of methods in the conformance check of method call. For instance, when methods specified in the INVOKE statement are not in a class or parameters to be passed to methods are different, they are checked.

```
    :
WORKING-STORAGE SECTION.
01  OBJ-1        USAGE OBJECT REFERENCE EMPLOYEE.
01  OBJ-2        USAGE OBJECT REFERENCE MANAGER.
    :
PROCEDURE DIVISION.
       :
    INVOKE EMPLOYEE "NEW" RETURNING OBJ-1.
    INVOKE OBJ-1 "CMPT-MPAY" RETURNING Special-allowance.  ... [1]
       :
    INVOKE MANAGER "NEW" RETURNING OBJ-2.
    INVOKE OBJ-2 "CMPT-MPAY" USING Special-allowance.      ... [2]
       :
```

Explanation of diagram

[1] is an error because a method that does not exist is specified.

[2] is an error because the parameter to the CMPT-MPAY method is different.

The following explains the concept of conformance with an example of the relationship between the employee class and manager class:

The manager class can be executed just like the employee class because it includes all functions of the employee class. In other words, it is possible to handle objects of the manager class with the interface of the employee class. On the other hand, the employee class does not include all functions of the manager class. Therefore, it is not possible to handle objects of the employee class using the interface of the manager class. To describe such a relationship, the word conformance was defined in the Object-Oriented programming.

13.4.2 Object Reference Types and Conformance Checking

There are different types of object reference definition items, different object reference data to be stored, and different ways to check for conformance.

Object reference data items include roughly the following 3 types and they are used distinctively to achieve polymorphism and dynamic binding. For more information, refer to "13.6.2 Dynamic Binding and Polymorphism".

```
  :
01  OBJ-1  USAGE OBJECT REFERENCE.                 ... [1]
01  OBJ-2  USAGE OBJECT REFERENCE EMPLOYEE.        ... [2]
01  OBJ-3  USAGE OBJECT REFERENCE EMPLOYEE ONLY.   ... [3]
  :
```

- [1] This definition type can store object references of any class. In this case, coding is easy (until the object program can be completed) because the conformance check is not carried out during compilation. Therefore, errors may be found in the conformance check during execution.
- [2] This definition type is used in the previously discussed examples and specifies explicitly that the object reference of the specified class (the employee class in the example) is stored. In this case, the object reference of the child class of employee class or the employee class (manager class) can be stored.
- [3] This definition type is used to store only object references of the specified classes. In this case, the object references of the class that conforms the specified class cannot be stored.

Note that definitions [2] and [3] have different specifications depending on factory object or object instance. In the example, the specification is that object references of object instances are stored. Define as follows for factory objects:

```
  :
01  OBJ-2  USAGE OBJECT REFERENCE FACTORY OF EMPLOYEE.
01  OBJ-3  USAGE OBJECT REFERENCE FACTORY OF EMPLOYEE ONLY.
  :
```

Object reference data items of factory objects have not been discussed. However, the usage is the same as the case of object instances, refer to the figure below:

```
   :
WORKING-STORAGE SECTION.
```

```
01  OBJ-F  USAGE OBJECT REFERENCE FACTORY OF EMPLOYEE.
01  OBJ-1  USAGE OBJECT REFERENCE EMPLOYEE.
   :
PROCEDURE DIVISION.
   :
    SET OBJ-F TO EMPLOYEE.
    INVOKE OBJ-F "NEW" RETURNING OBJ-1.
   :
```

13.4.3 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 executing conformance checks.

13.4.3.1 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.
01  OBJ-3       USAGE OBJECT REFERENCE EMPLOYEE ONLY.
  :
01  OBJ-X       USAGE OBJECT REFERENCE.
01  OBJ-Y       USAGE OBJECT REFERENCE MANAGER.
01  OBJ-Z       USAGE OBJECT REFERENCE MANAGER 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 errors do 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 errors (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 errors occurs in [7], [8], and [9]. A conformance check is carried out during compilation.

13.4.3.2 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.
01  OBJ-3       USAGE OBJECT REFERENCE EMPLOYEE ONLY.
```

```
  :
PROCEDURE DIVISION.
  :
    INVOKE OBJ-1 "CMPT-SAL" RETURNING SALARY.      ... [1]
    INVOKE OBJ-2 "CMPT-SAL" RETURNING SALARY.      ... [2]
    INVOKE OBJ-3 "CMPT-SAL" RETURNING SALARY.      ... [3]
  :
```

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.

13.5 Repository

The file called "Repository File" (class-name.rep) is created when a class definition is compiled. Repository files act as input to a compiler when the program or class that uses the class is compiled and used for the conformance check.

This section explains the overview of repository file. For more details, refer to "15.6.1 Repository File and Compilation Procedure".

13.5.1 Overview of Repositories

Repositories are files that contain information about classes. In Object-Oriented applications, you will mostly be concerned with the class repository. The class repository is a file created when a class is compiled.

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.

13.5.1.1 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.

<Manager class>

```
IDENTIFICATION DIVISION.
 CLASS-ID. MANAGER INHERITS EMPLOYEE.
 ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    CLASS EMPLOYEE.
 :
```

The parent class (class specified in the INHERITS clause) must be specified in the repository paragraph.

Information

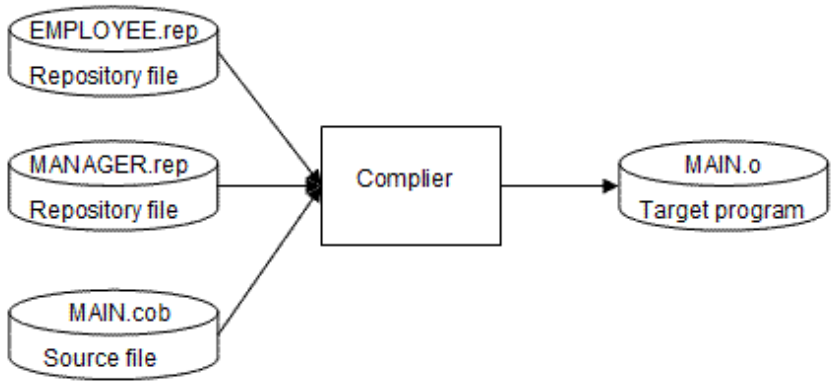
A structure of two or more hierarchies to be inherited can be compiled by inputting the repository file of the parent class directly. For example, when the manager class is compiled, the FJBASE class is inherited indirectly; however, the repository file of the FJBASE class does not need to be input.

13.5.1.2 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. MAINPGM.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    CLASS EMPLOYEE
    CLASS MANAGER.
     :
DATA DIVISION.
WORKING-STORAGE SECTION.
01  OBJ-1       USAGE OBJECT REFERENCE EMPLOYEE.
01  OBJ-2       USAGE OBJECT REFERENCE MANAGER.
    :
PROCEDURE DIVISION.
      :
    INVOKE EMPLOYEE "NEW" RETURNING OBJ-1.
    INVOKE OBJ-1 "SET-EMPL" USING EMPLOYEE-DATA.
      :
    INVOKE MANAGER "NEW" RETURNING OBJ-2.
    INVOKE OBJ-2 "SET-EMPL" USING EMPLOYEE-DATA.
      :
```

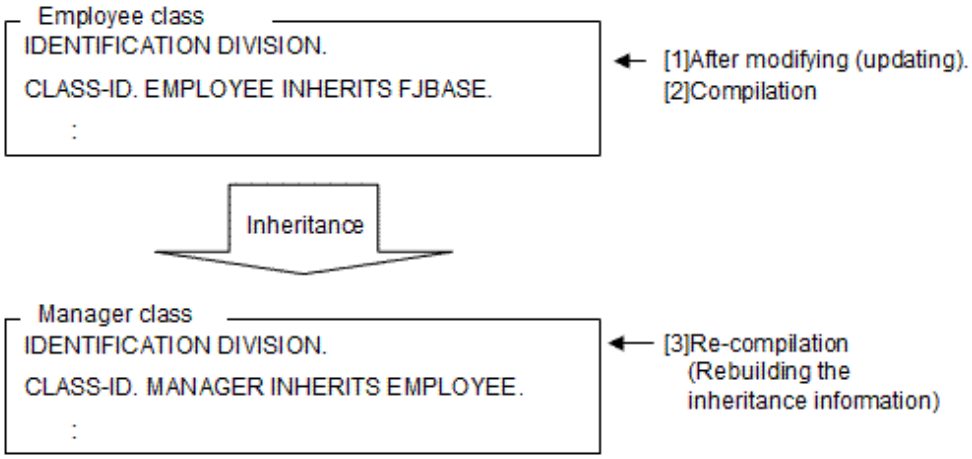


13.5.2 Effects of Updating Repository Files

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.



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 MANAGER.
*> :
DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  OBJ-1 USAGE OBJECT REFERENCE.
 01  OBJ-2 USAGE OBJECT REFERENCE.
*> :
PROCEDURE DIVISION.
*>      :
    INVOKE EMPLOYEE "NEW" RETURNING OBJ-1.  *>--+ For reattempting of
*>      :                                   *>  |  the conformance check
    INVOKE MANAGER "NEW" RETURNING OBJ-2.   *>--+
```

Users should execute these recompilation. Therefore, pay much attention when you correct the class definition that has been constructed once. You can easily correct using the project management function because the necessary recompile is executed only by registering the dependency relationship beforehand.

For the information on project management function, refer to "Chapter 15 Developing and Executing Object-Oriented Programs".

13.6 Method Binding

Binding is the process by which names in code are connected to actual data or procedures. For methods, the Object-Oriented 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. In the Object-Oriented, use the term "Method Binding" for determining the method to be invoked.

This section explains the method binding.

13.6.1 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:

```
*>    :
 WORKING-STORAGE SECTION.
 01  OBJ-1      USAGE OBJECT REFERENCE EMPLOYEE ONLY.
 01  SALARY     PIC 9(08).
*>    :
 PROCEDURE DIVISION.
*>    :
     INVOKE EMPLOYEE "NEW" RETURNING OBJ-1.        *> [1]
*>    :
     INVOKE OBJ-1 "CMPT-SAL" RETURNING SALARY.     *> [2]
*>    :
     INVOKE SUPER "CMPT-SAL" RETURNING SALARY.     *> [3]
*>    :
```

Explanation of diagram

- [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. For more details, refer to "13.6.3 Binding with the Predefined Object Identifier SUPER".

13.6.2 Dynamic Binding and Polymorphism

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:

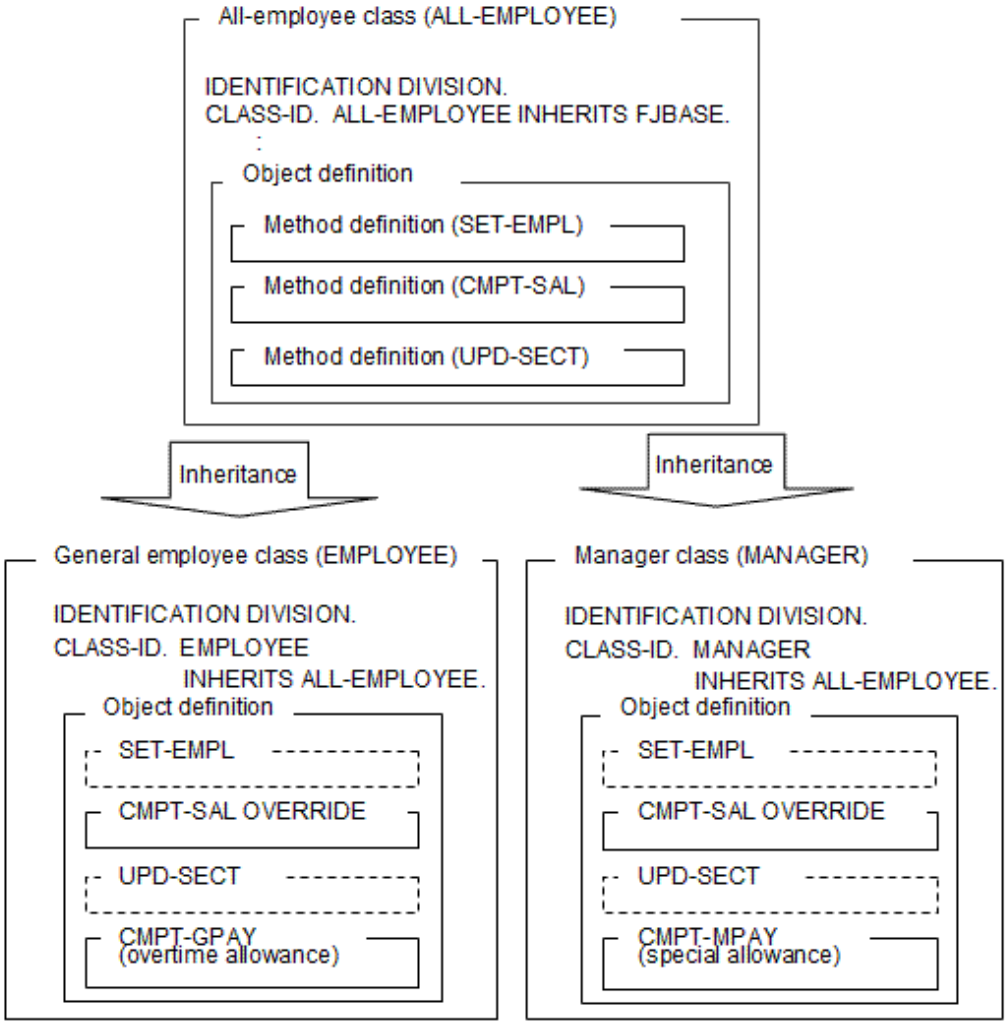
```
*>    :
 WORKING-STORAGE SECTION.
 01  OBJ-1       USAGE OBJECT REFERENCE.
 01  OBJ-2       USAGE OBJECT REFERENCE EMPLOYEE.
*>    :
 PROCEDURE DIVISION.
*>    :
     INVOKE OBJ-1 "CMPT-SAL"  RETURNING SALARY.    *> [1]
*>    :
     INVOKE OBJ-2 "CMPT-SAL"  RETURNING SALARY.    *> [2]
*>    :
     INVOKE SELF  "CMPT-SAL"  RETURNING SALARY.    *> [3]
*>    :
```

Explanation of diagram

- [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. For more details, refer to "13.6.4 Binding with the Predefined Object Identifier SELF".

Dynamic binding enables the same code to invoke different routines at execution time. This function is called Polymorphism

For example, you could structure the 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:



[Employee management program]

DATA DIVISION

```
  WORKING-STORAGE SECTION.
  01  OBJ-1   USAGE OBJECT REFERENCE EMPLOYEE.
  01  OBJ-2   USAGE OBJECT REFERENCE MANAGER.
  01  OBJ-X   USAGE OBJECT REFERENCE All-EMPLOYEE.
*>   :
```

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.
```

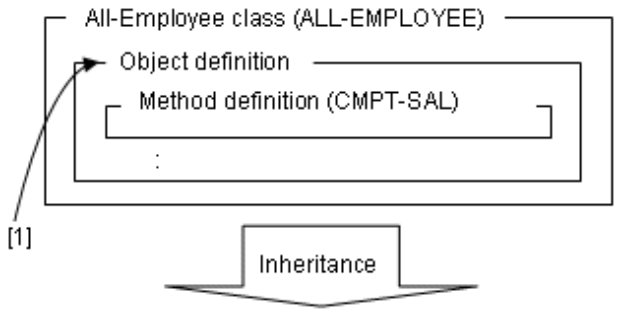
```
     INVOKE OBJ-X "CMPT-SAL" RETURNING SALARY.
*>   :
```

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.

13.6.3 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:



[Manager class]

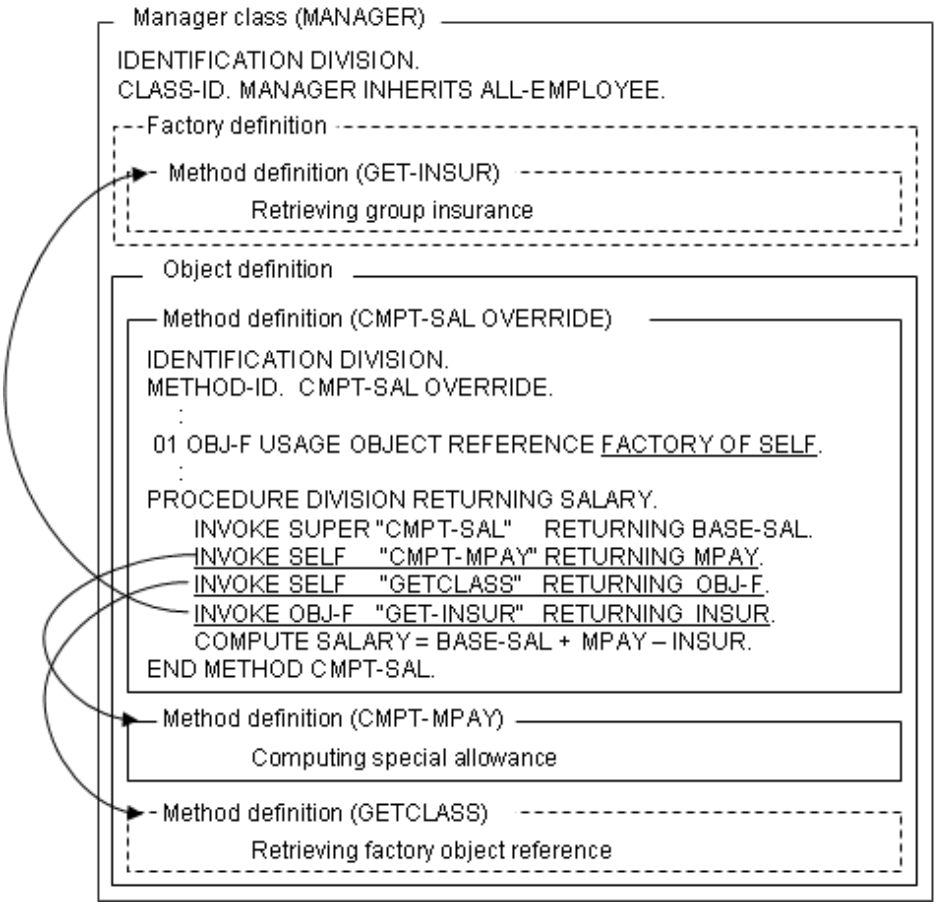
```
CLASS-ID. MANAGER INHERITS ALL-EMPLOYEE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS ALL-EMPLOYEE.
 FACTORY.
 PROCEDURE DIVISION.
 METHOD-ID. GET-INSUR.
 DATA DIVISION.
  LINKAGE SECTION.
  01 INSUR     PIC 9(08).
 PROCEDURE DIVISION RETURNING INSUR.
     MOVE 500 TO INSUR.
 END METHOD GET-INSUR.
 END FACTORY.
 OBJECT.
 PROCEDURE DIVISION.
 METHOD-ID. CMPT-SAL 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 "CMPT-SAL"  RETURNING BASE-SAL.  *>--->[1]
     INVOKE SELF  "CMPT-MPAY" RETURNING MPAY.
     INVOKE SELF  "GETCLASS"  RETURNING OBJ-F.
     INVOKE OBJ-F "GET-INSUR" RETURNING INSUR.
     COMPUTE SALARY = BASE-SAL + MPAY - INSUR.
 END METHOD CMPT-SAL.
```

Thus, the object unique name SUPER that has been defined is used to express the parent class.

13.6.4 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 CMPT-SAL method for the all-employee class, but you want a CMPT-MPAY method to work for both manager and all-employee classes. You could code the following:



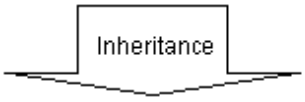
Care must be taken if you use the predefined object unique name SELF because methods to be called are always decided in the runtime (dynamic binding). That is, the methods to be called by the object under execution are changed when the overwritten method exists.

For instance, in the above-motioned example, assume there is a method of calculating bonus in all employee class and processing for calculating the salary in the method would be needed. Then predefined object unique name SELF would allow you to describe as follows:

[All-Employee class]

```
CLASS-ID. ALL-EMPLOYEE INHERITS FJBASE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS FJBASE.
 OBJECT.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*>    :
 PROCEDURE DIVISION.
*>---------------------------------
*>[1] Salary Calculation Method
*>---------------------------------
 METHOD-ID. CMPT-SAL.
 DATA DIVISION.
```

```
  LINKAGE SECTION.
  01  SALARY  PIC 9(08).
 PROCEDURE DIVISION RETURNING SALARY.
     MOVE BASIC-SALARY TO SALARY.
 END METHOD CMPT-SAL.
*>---------------------------------
*>[2] Bonus Calculation Method
*>---------------------------------
 METHOD-ID. CMPT-BONUS.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  SALARY  PIC 9(08).
 PROCEDURE DIVISION.
*>      :
     INVOKE SELF "CMPT-SAL" RETURNING SALARY.  *>[3]  invoking[1]
     DISPLAY "SALARY =" SALARY
*>      :
 END METHOD CMPT-BONUS.
*>      :
```



[Manager class]

```
CLASS-ID. MANAGER INHERITS ALL-EMPLOYEE.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS ALL-EMPLOYEE.
 OBJECT.
 PROCEDURE DIVISION.
*>---------------------------------
*>[4] Salary Calculation Method
*>---------------------------------
 METHOD-ID. CMPT-SAL OVERRIDE.
*> :
 END METHOD Compute-Salary.
*>
*>---------------------------------
*>[5] Bonus Calculation Method
*>---------------------------------
*> The CMPT-BONUS method was the one defined implicitly
*> by inheriting the ALL-EMPLOYEE class.
*>
*> METHOD-ID. CMPT-BONUS.
*> DATA DIVISION.
*>  WORKING-STORAGE SECTION.
*>  01  SALARY  PIC 9(08).
*> PROCEDURE DIVISION.
*>      :
*>     INVOKE SELF "CMPT-SAL" RETURNING SALARY.  *>[6] invoking [4]
*>     DISPLAY "SALARY =" SALARY
*>      :
*> END METHOD CMPT-BONUS.
 END OBJECT.
 END CLASS MANAGER.
```

Explanation of figure

When the method of calculating the bonus ([2]) is called by the object reference of the employee class, the method of calculating the salary of [1] is called by the INVOKE statement of [3]. However, when the method of calculating the bonus ([5] implicit definition method) is

called by the object reference of the manager class, the method of calculating the salary of [4] (method of overwriting) is called by the INVOKE statement of [6]. In short, the salary for each class can be calculated in a suitable way. This is case of using Polymorphism.

13.7 Using More Advanced Functions

The basic concepts and functions of object-oriented programming have already been explained in the preceding sections. That is, object-oriented programming can be implemented using these functions. However, there are many other functions such as more advanced functions and those that make it easy to use (describe) the basic functions.

This section explains how to use the extended functions.

13.7.1 PROTOTYPE Declaration of a Method

Usually, the method definition described in the class definition can 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 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.       *>   [1]
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS ALL-EMPLOYEE.                   *>   [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.
```

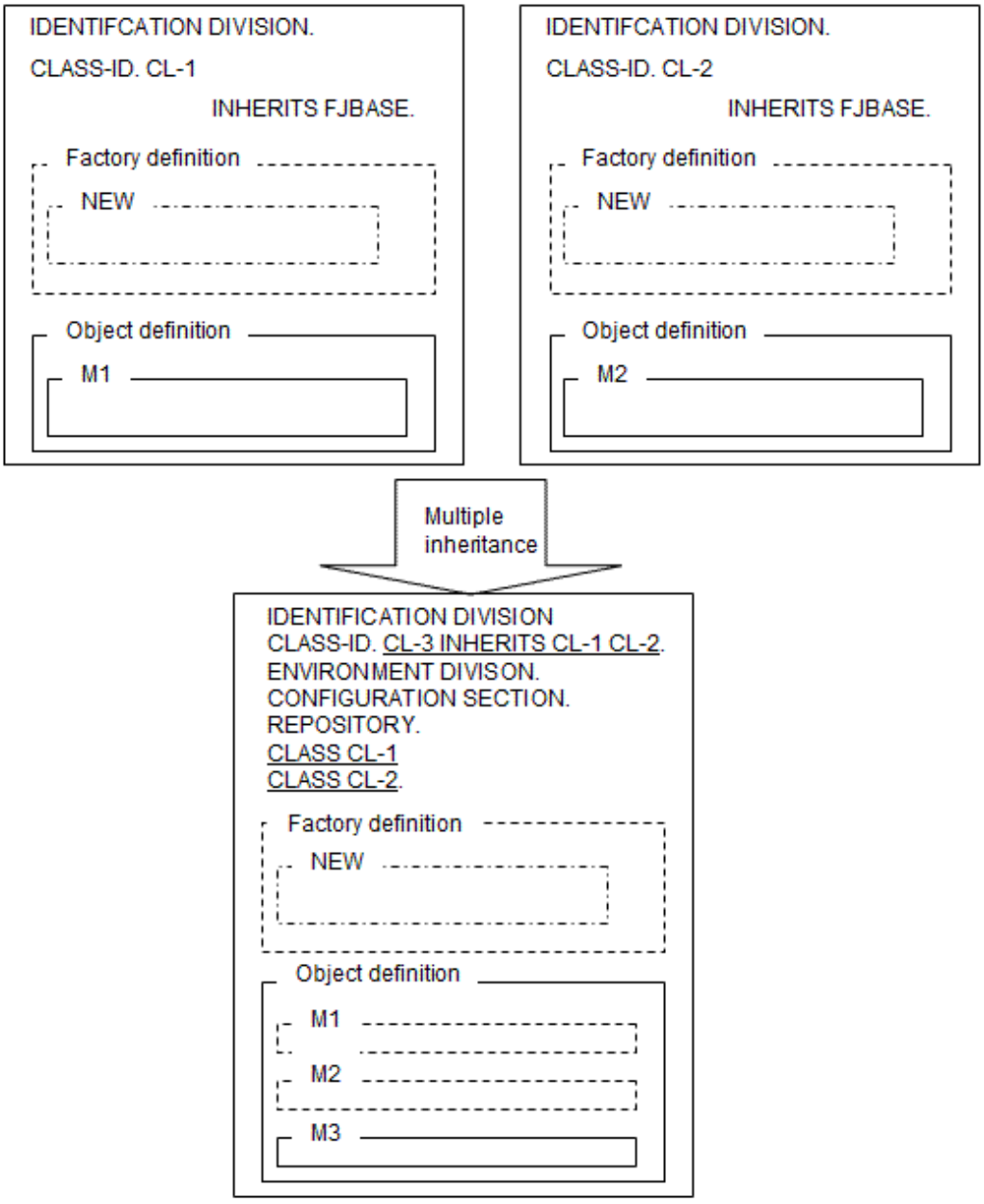
- [1],[2] Specify the name of a class logically containing the method.
- [3] Define data to be used in the method,
- [4] Describe a method procedure.

When two or more persons are developing a class definition, describe its method definition in separate compilation units as shown above.

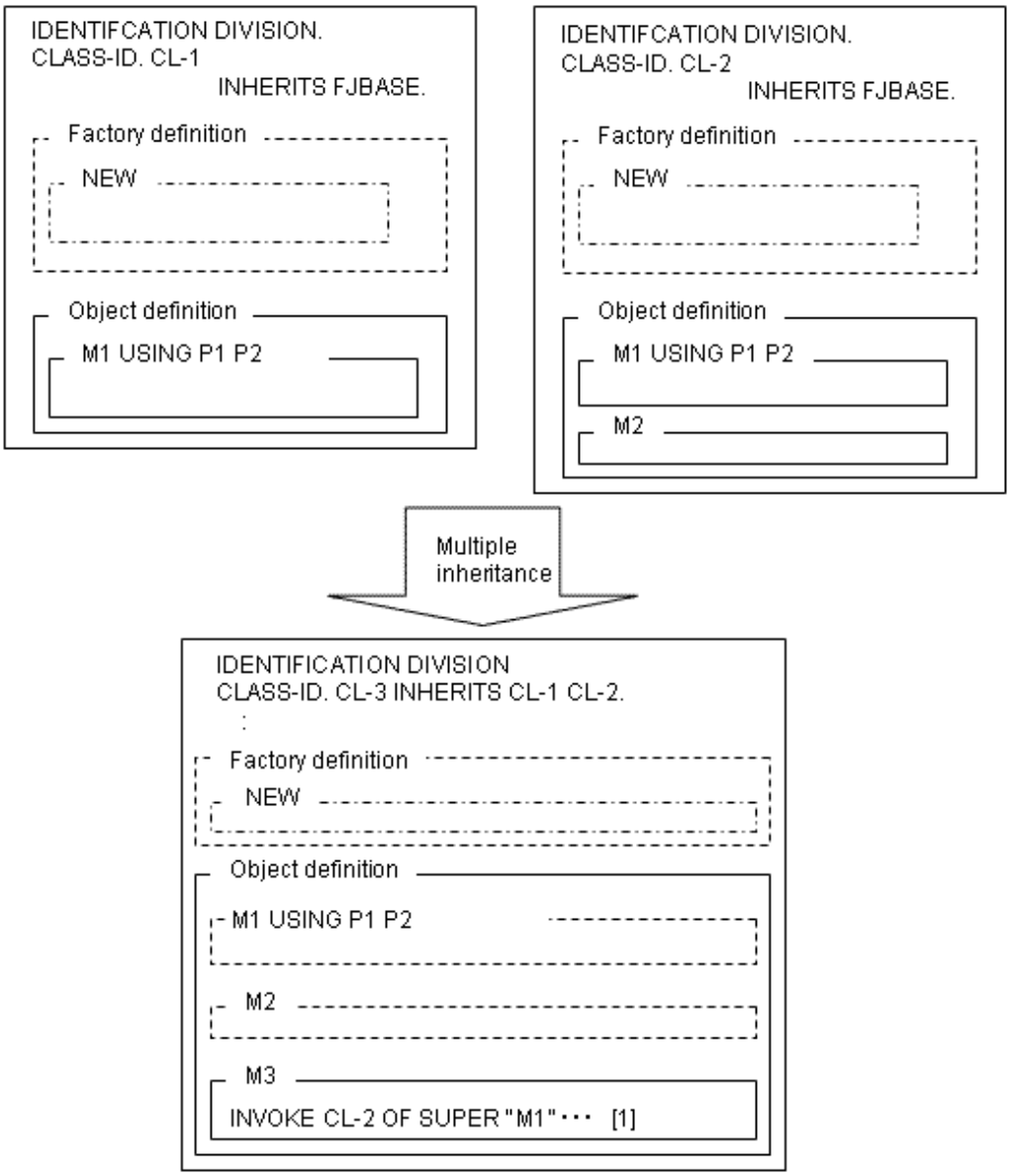
When compiling the separate method, input the repository file of a class logically containing that method. Therefore, the class definition must be compiled before the method is.

13.7.2 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 inheritances.



The logic of multiple inheritances is the same as that of single-class inheritance. However, 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).



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.

13.7.3 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.

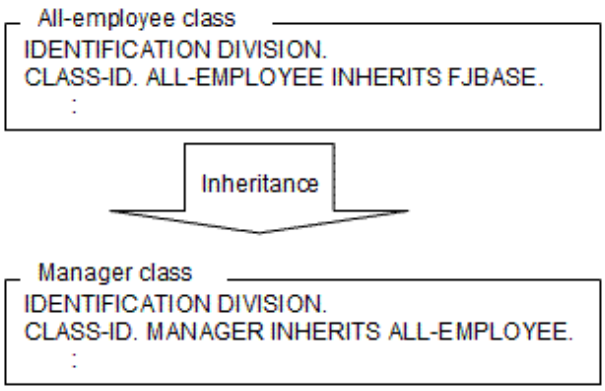
```
*>    :
 01  OBJ-1 USAGE OBJECT REFERENCE CL-1.
 PROCEDURE DIVISION.
*>    :
     INVOKE CL-1  "NEW" RETURNING OBJ-1
     INVOKE OBJ-1 "M1" USING P1 P2 RETURNING P3.
     IF P3 = 0 THEN
*>    :
```

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
*>    :
```

13.7.4 Object Specifiers

Conformance checks were 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.



When the inheritance relationship shown above exists, the assignment can be performed normally in [1] because the manager class is the child class of the all-employee class.

Employee management program:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MAIN.
 ENVIRONMENT DIVISION.
```

```
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS ALL-EMPLOYEE
      CLASS MANAGER.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  OBJ-1   USAGE OBJECT REFERENCE ALL-EMPLOYEE.
  01  OBJ-2   USAGE OBJECT REFERENCE MANAGER.
 PROCEDURE DIVISION.
*>    :
     INVOKE MANAGER "NEW" RETURNING OBJ-2.
*>    :
     SET OBJ-1 TO OBJ-2.  *>[1]
     SET OBJ-2 TO OBJ-1.  *>[2] error
*>    :
```

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 specifies to enable the assignment shown above.

Employee management program:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MAIN.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS ALL-EMPLOYEE
      CLASS MANAGER.
 DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  OBJ-1   USAGE OBJECT REFERENCE ALL-EMPLOYEE.
  01  OBJ-2   USAGE OBJECT REFERENCE MANAGER.
 PROCEDURE DIVISION.
*>    :
     INVOKE MANAGER "NEW" RETURNING OBJ-2.
*>    :
     SET OBJ-1 TO OBJ-2.
     SET OBJ-2 TO OBJ-1 AS MANAGER.
*>    :
```

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.

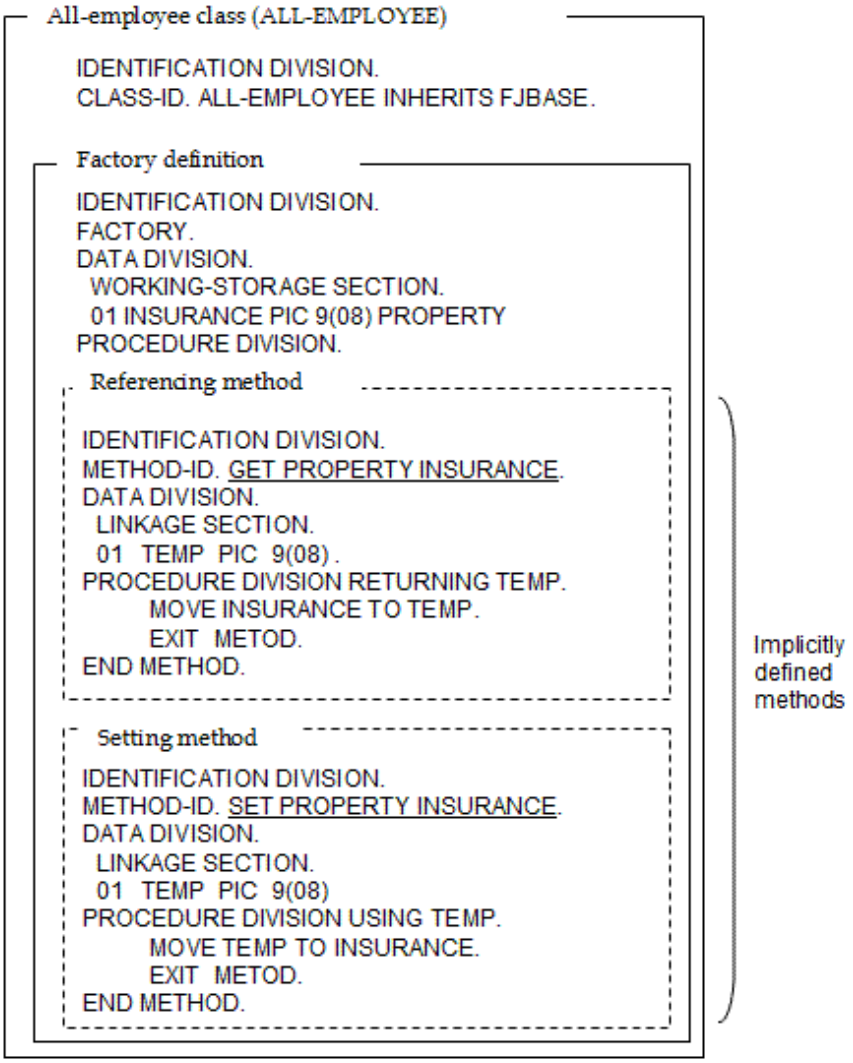
When an object specifier is used, a conformance check is made with a specified class name during compilation. During execution, however, the conformance check is made with an object reference actually stored. That is, an error is assumed at execution time if a conformance rule is infringed upon.

13.7.5 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.



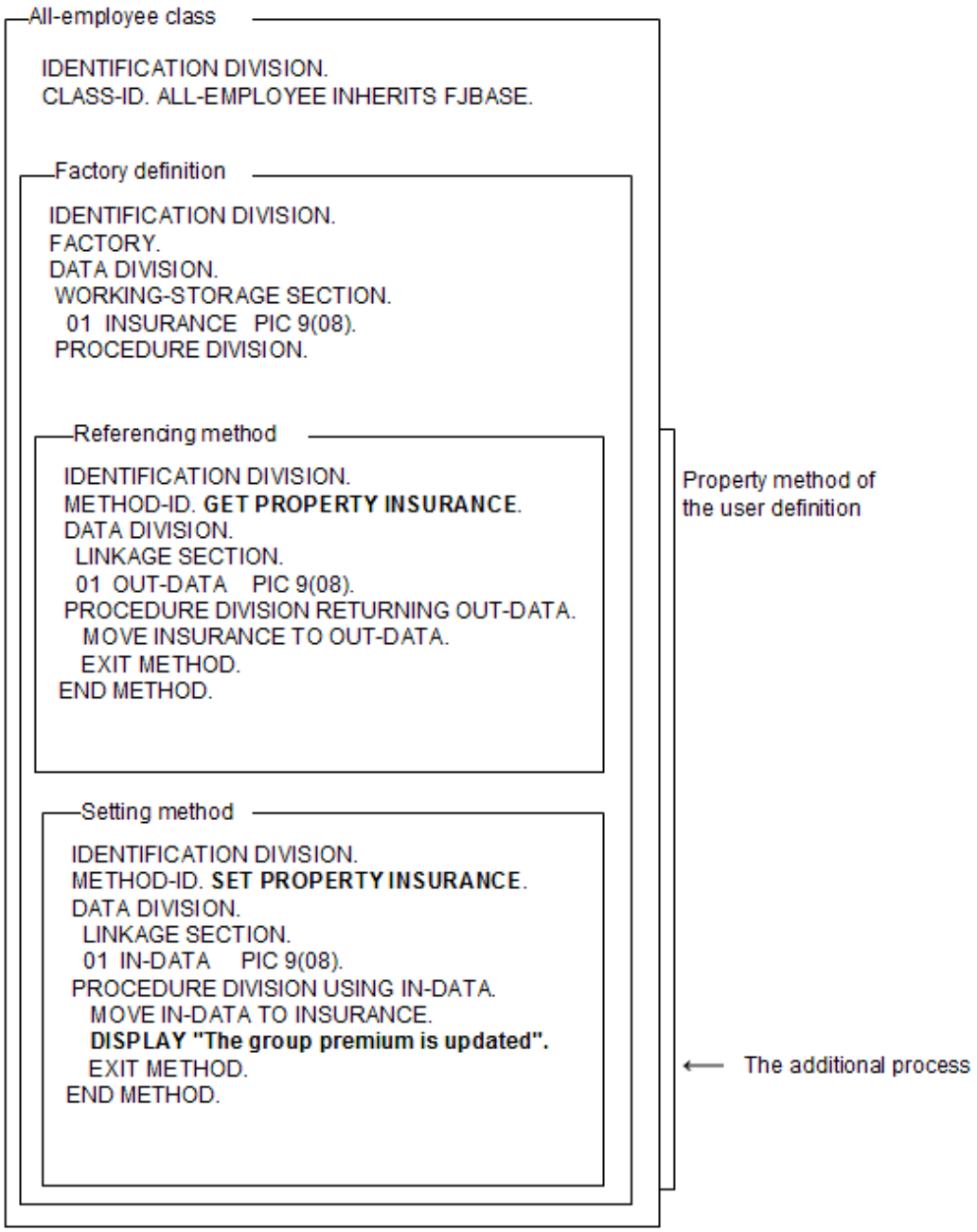
The method defined implicitly in the PROPERTY clause as shown above is referred to as a property method.

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. MAINPGM.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS ALL-EMPLOYEE
      CLASS MANAGER.
 DATA DIVISION.
*> :
 PROCEDURE DIVISION.
*>      :
*> GROUP INSURANCE PREMIUM SETTING PROCESS
*>
     MOVE insurance-premium TO INSURANCE OF ALL-EMPLOYEE.  *> invokes the setting
     MOVE insurance-premium TO INSURANCE OF MANAGER.       *> method.
*>
*> GROUP INSURANCE PREMIUN RETRIEVAL PROCESS
*>
   Group-insurance-process.
     EVALUATE Employee-id
```

```
     WHEN id-Employee
       MOVE INSURANCE OF ALL-EMPLOYEE TO insurance-premium *> invokes the
     WHEN id-Manage                                        *> referencing method.
       MOVE INSURANCE OF MANAGER 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.



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.

13.7.6 Initialization and Termination Methods

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 "13.2.2 Object Life").

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.

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 "13.3.3 Overriding 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

When using _FINALIZE, do not describe the STOP RUN statement in the _FINALIZE method. Also, do not describe the STOP RUN statement in the programs and methods to be invoked from _FINALIZE.

13.7.7 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 the above figure, 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"  *> --+ Deploy the nest 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.

13.7.8 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 executable file.

13.7.8.1 Cross Reference Patterns

The cross reference patterns are classified into the following three types:

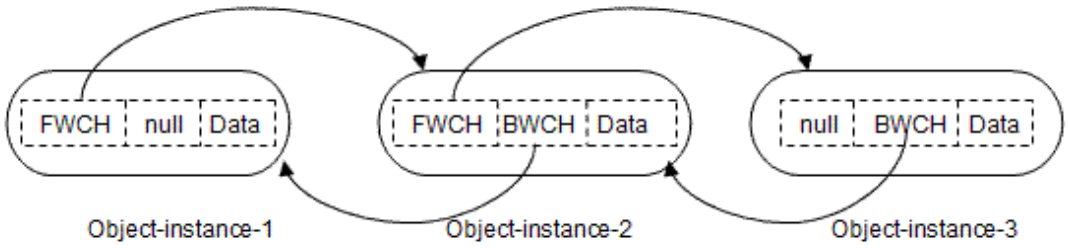
- Cross reference by the self class.
- Direct cross reference with the other class.
- Indirect cross reference with the other class.

Each pattern is explained 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.
*>     :
```



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

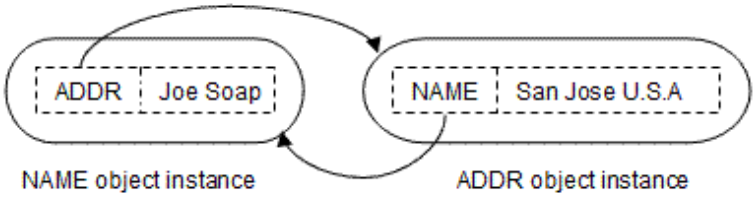
When an object instance has a close relationship with that of the other class, that is, when it can be followed from another, a direct cross relationship is established. This type of definition is explained below using NAME and address ADDR classes as examples.

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.
*>     :
```



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.
*>     :
```

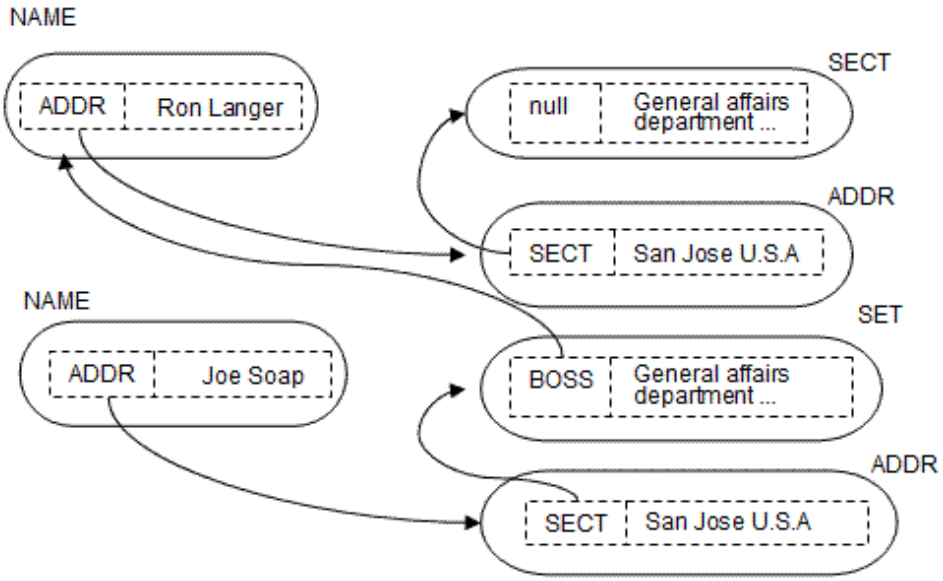
Address class

```
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.
*>     :
```



When object instances are linked in a complicated manner as shown above, an indirect cross reference relationship is established with ease.

13.7.8.2 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.

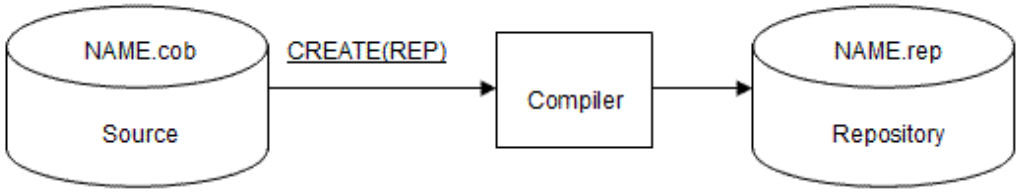
An example of direct cross reference (NAME and ADDR classes) is explained below.

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.

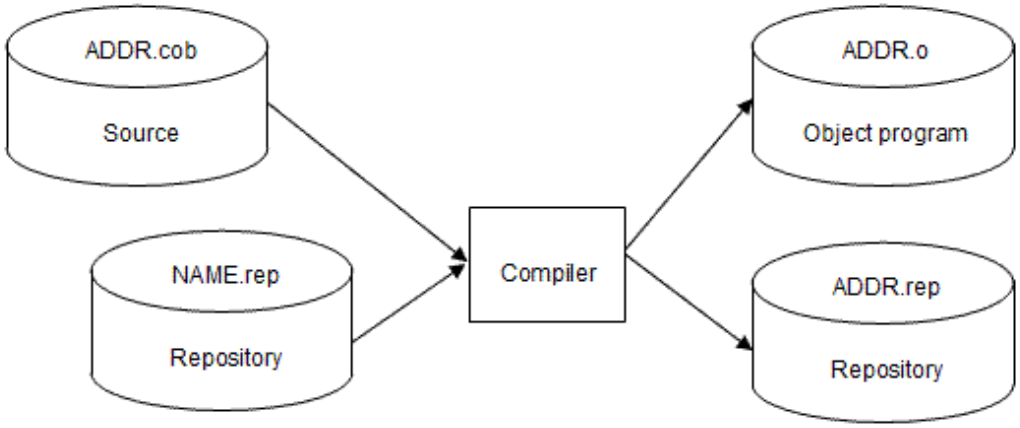
Step 1: Creating the repository file of the name class

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.



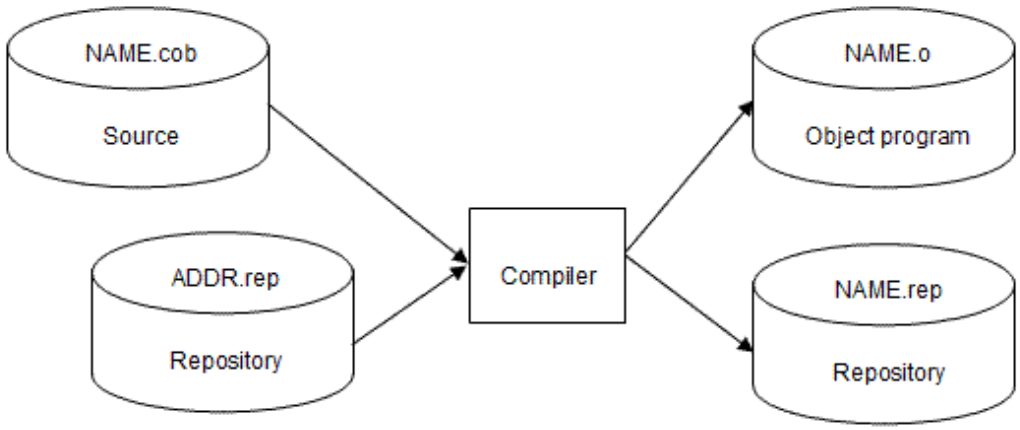
Step 2: Compiling the address class using the created repository

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.

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.

13.7.8.3 Linking Cross Reference Classes

When creating the executable file 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 shared object file, no special considerations are required. When they are composed by each shared object file, special considerations are required.)

The following explains an example of the direct cross reference (name and address classes).

The shared object file of the address class is required to link the shared object of the name class. To create the shared object file of the address class, the address class must be linked. In this case, however, the shared object file of the name class is required. That is thrown into the state of dead lock as in the case of compile time.

To prevent situations like this, do not link classes in the cross reference relationship when shared object files of each class are created (step 1). Classes with the mutual reference relationship at the stage where executable files are linked should be linked (step 2-1). When an executable file is not created, the shared object files created in step 1 should be re-linked. Re-link the classes in the cross reference relationship (step 2-2).

Step 1: Creating the shared object file of the name and address class

Create the shared object file of the name and address class. At this time, the cross reference classes are linked.

```
$  cobol -dy -shared -o libNAME.so NAME.o
$  cobol -dy -shared -o libADDR.so ADDR.o
```

Step 2-1: Creating executable file

Create an executable file. Link shared object files of the name class and address class.

```
$ cobol -o sample -L. -lNAME -lADDR sample.o
```

Step 2-2: When you do not create an executable file

When an executable file is not created, use and re-link shared object files of the name class and address class created in step 1.

```
$  cobol -dy -shared -o libNAME.so -L. -lADDR NAME.o
$  cobol -dy -shared -o libADDR.so -L. -lNAME ADDR.o
```

13.7.8.4 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 14 Advanced Features of Objected-Oriented COBOL

This chapter provides details of how to use some of the more advanced features of Objected-Oriented COBOL.

14.1 Exception Handling

This section is an overview and explanation of how to use the exception process.

14.1.1 Overview

The exception procedure can be specified by describing the USE statement in the DECLARATIVES in the PROCEDURE DIVISION. When exception condition is occurred, the exception procedure is executed. The exception condition that may be generated includes exception objects.

14.1.2 Exception Object

The exception object is used to wrap up error handling in one place. For instance, assume that you execute processing to display a message if invalid data has been input.

A class (DATA-ERROR) with procedures by which a data error message is displayed is defined. When an error is found in data, it is objectified and the object is specified for the RAISE statement or the EXIT statement with RAISING phrase.

Objects specified then are referred to as exception objects with the exception condition "An error occurred in data".

When an exception object is raised and there is use statement for which specified a class name in inheritance relationship with the class of exception object, the exception procedure of its use statement is executed. In the above example, the procedure is executed when DATA-ERROR is described in the USE statement. To use an exception object in an exception procedure, you can refer it as EXCEPTION-OBJECT. When the exception procedure terminates normally, the control return immediately after where the exception condition is raised.

Wherever and whenever the exception is occurred, the exception procedure is executed. Therefore, you can describe the procedure for data input error wrapping into one place.

- Specify object instances for exception objects.
- Declare class names that are specified for the USE statement in REPOSITORY paragraph.
- When 2 or more USE statements for an exception object are described in the DECLARATIVES, the exception procedure of the head is executed.
- When the following exception conditions are occurred, the exception process of [1] is executed.
  - The object of CLASS-A is specified as an exception object.
  - CLASS-A inherits CLASS-B and CLASS-C.

```
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.
```

- In the following cases,

  a. There is no exception procedure.

  b. The exception procedure cannot be executed such as the exception object is a NULL object.

  the statements that raise the exception are executed as follows:

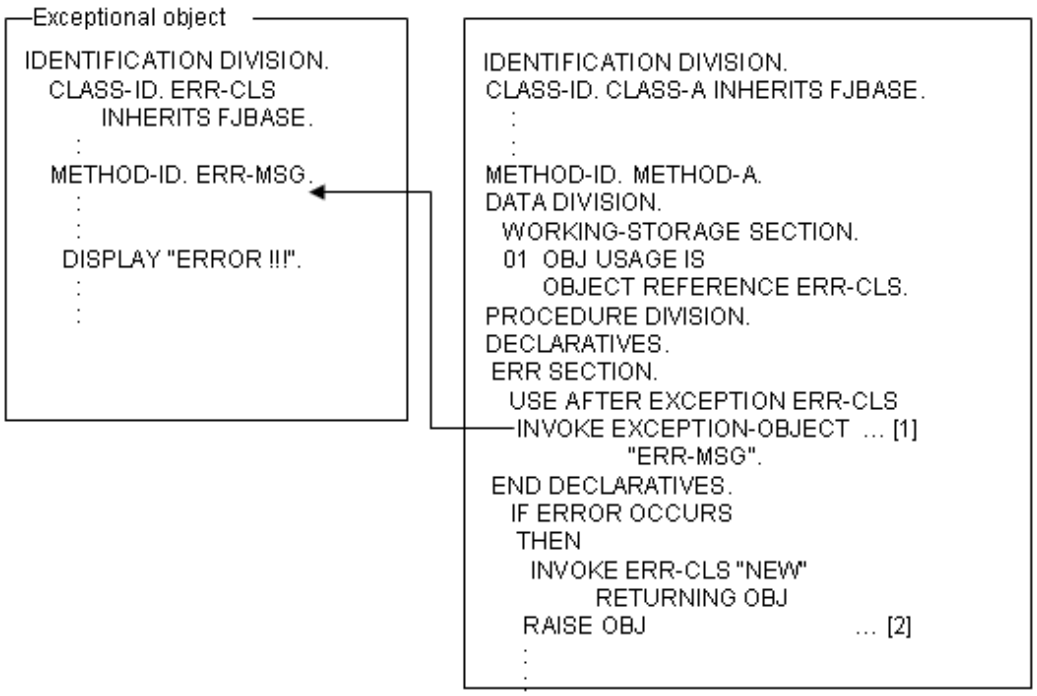
  - RAISE statement : The control moves to the statement immediately after the RAISE statement.
  - The EXIT statement with a RAISING phrase : Program or method is terminated abnormally.

- During execution of a specific USE procedure due to an exception, another exception may occur and cause transfer to the same USE procedure, so the USE procedure would be recursively called. In such cases, if execution control reaches the end of the USE procedure, the program or method terminates abnormally.

14.1.3 The RAISE Statement

When you recognize an error condition in your program and want to invoke the general error handling routine through exception handling, 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:

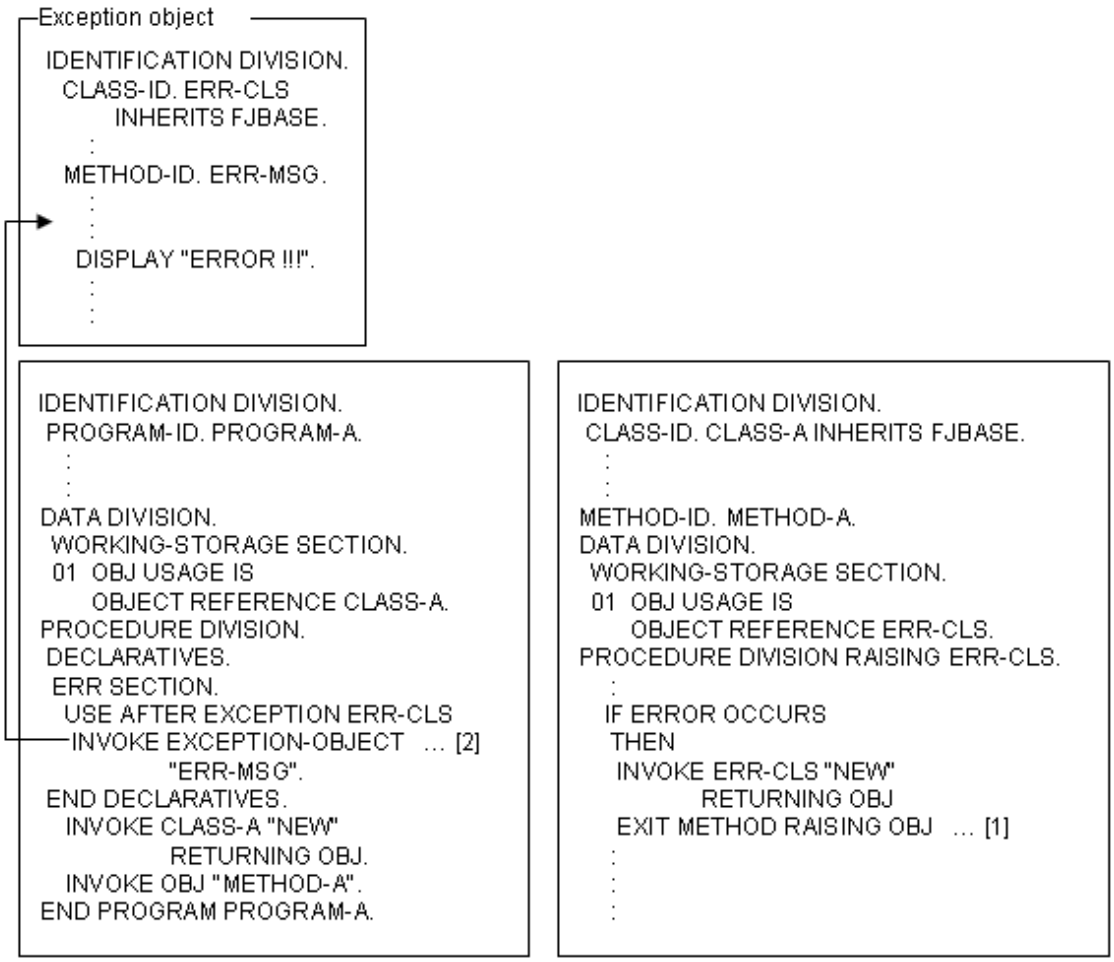


Explanation of diagram

After executing the RAISE statement [2], the procedure (INVOKE statement [1]) for occurred exception condition is executed.

14.1.4 The EXIT Statement with RAISING Phrase

The EXIT statement with a RAISING phrase has the effect of exiting the program or method, returning to the invoking code, and then occurring the exception condition. The system performs the exception procedure described in the invoking code or method.



Explanation of diagram

When the EXIT statement with the RAISING phrase is executed by a called program or method ([1]), the exception procedure (INVOKE statement of [2]) is executed after returning to the called object.

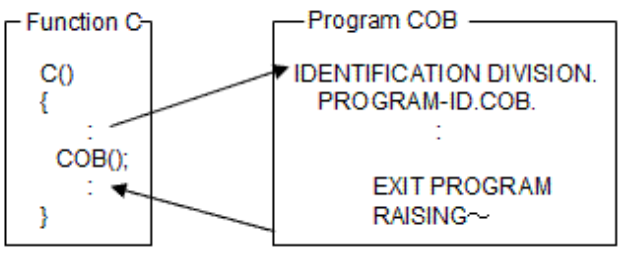
Note

In the case of the following types of COBOL programs, the exception condition cannot be occurred by EXIT statement:

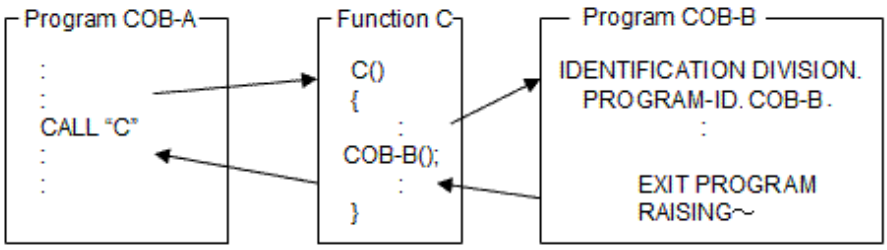
- Main program.
- COBOL program invoked from a program written in another language (refer to the following figure [1]).

However, exception condition can be occurred by the EXIT statement at COBOL program (refer to the following figure [2]) called from COBOL program via other programming language program.

[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:



14.2 Using C++ with Object-Oriented COBOL

This section explains how to use C++ programs with NetCOBOL Object-Oriented COBOL.

14.2.1 Overview

This section explains how to use C++ programs with COBOL language. When linking with C++ programs, knowledge from "8.3 Linkage with C Language Program" is used as a foundation because the calls of C or C++ functions are internally used.

14.2.2 How to Use C++ Objects

C++ is widely used as an Object-Oriented programming language and a lot of useful classes are defined. Classes defined in C++ cannot be used directly from COBOL because of the difference of the structures of the object codes. However, C++ programs can be executed by using the following 2 methods:

Method-1: Only the results from operating objects is used by defining the functions that execute objects in the C++ side and linking programs of C or C++ from COBOL.

Method-2: Defining interface programs in the C++ side and COBOL side allows you to operate as objects from COBOL.

Method01 can be achieved by a simple external program call. When you need only results of object executions, this way is sufficient. Method-2 allows you to execute classes of C++ just like COBOL classes. In other words, the member function of C++ is called by the INVOKE statement of COBOL, which enables to refer to/set member variables by the statement structure of reference/set of property.

The following section explains about Method-2:

14.2.3 Overview of Using C++ Objects

This section explains the overview of the using C++ objects.

14.2.3.1 Corresponding Classes in COBOL and C

In the object handling of C++, you need not consider the member function/variable that is not public of the class. Also, you do not have to consider classes that have been inherited and defined. Only interfaces that the classes display matters. Objects of COBOL and C++ objects correspond on this viewpoint as in "Table 14.1 Correspondence of COBOL classes and C++".

Table 14.1 Correspondence of COBOL classes and C++

Concept	C++	COBOL
Class	Class	Class
Object	Object	Object (instance)
Object data	Member variable declared as public	Property
Method	Member function declared as public	Method

14.2.3.2 The overview of mechanism

To use C++ objects, create interface programs in COBOL and C++.

COBOL Environment

An interface class is created that lets COBOL programs and methods access the C++ class as a COBOL class.

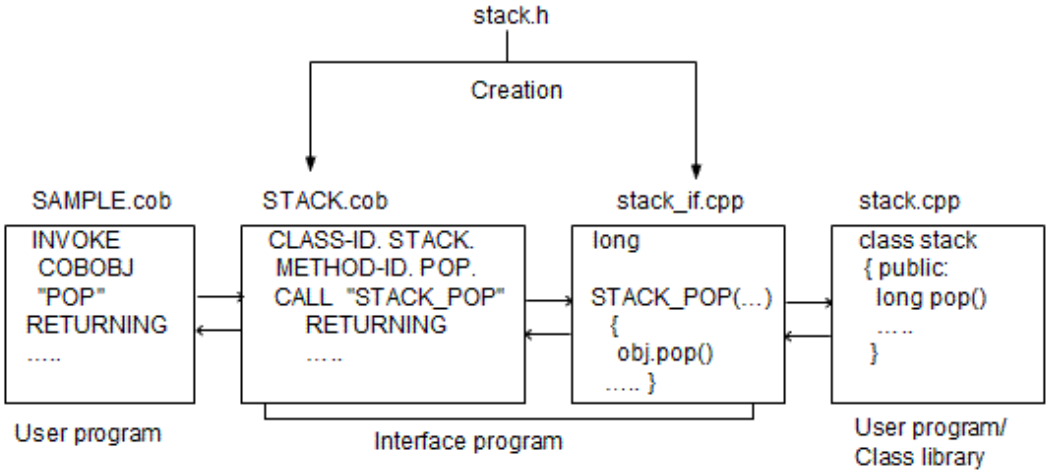
C++ Environment

Define functions to execute classes and objects to be called from the COBOL interface class.

Interface functions are created to access the C++ class public functions and data. Because C++ classes are usually defined in header files, the interface programs can be created by inspecting the header files.

"Figure 14.1 Image of interface program" shows the image of an interface program. "STACK.cob" and "Stack_if.cpp" are created from class definition (stack.h). The INVOKE statement of COBOL program relays the interface programs of COBOL and C++ and become a call of the member function of the class finally defined in C++.

Figure 14.1 Image of interface program



14.2.3.3 The Mechanism of Interface Programs

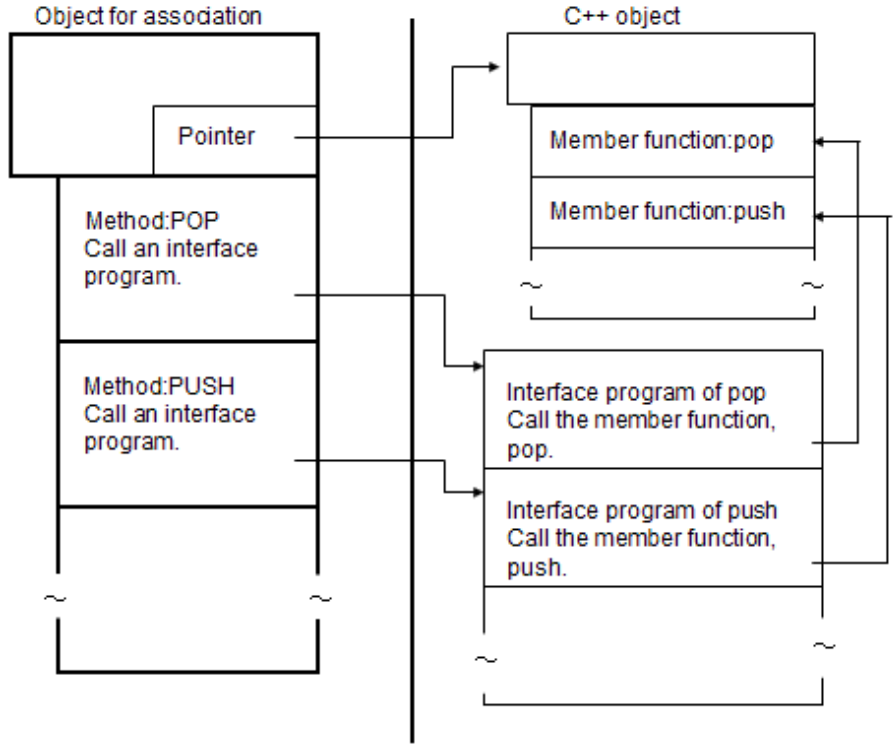
The following explains the mechanism of executing C++ objects programs.

Collaborating Program Structure

- Create a COBOL class with the same structure as the target C++ class.
- In the COBOL object data include an item to hold a pointer to the C++ object. This pointer is passed to the C++ interface program along with other required arguments.

- The C++ interface programs call the member functions in the corresponding objects.

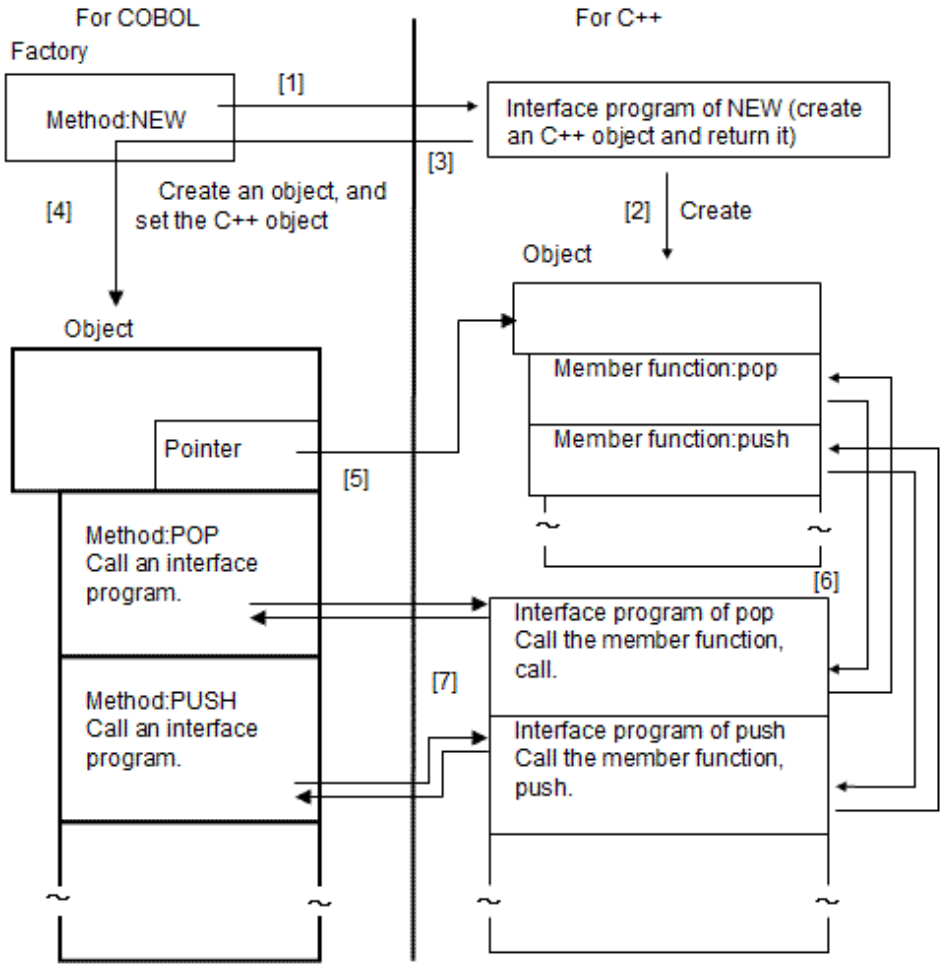
Figure 14.2 Interface program structure



Runtime Behavior

Flow of control in runtime is shown in Figure below.

Figure 14.3 Flow of control in runtime



Explanation of diagram

- [1] When the NEW method in the COBOL interface class is invoked, it invokes the NEW function of the C++ interface program.
- [2] The C++ interface program creates the C++ object.
- [3] The 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.

14.2.4 Steps for a Program with C

To create the interface programs you go through the following steps:

1. Check the C++ class definition.
2. Define the COBOL class.
3. Define the C++ interface programs.

These steps are detailed below based on this C++ class definition:

```
class stack {
public:
unsigned long pntr;
long pop();
void push(long val);
private:
long data[100];
} ;
```

14.2.4.1 Check the C++ class definition

Check the C++ to be operated by COBOL first. From the C++ class definition select the member functions and member variables that you wish to make available to the COBOL environment.

The following example makes all the member functions and variables available:

- Member Functions
  - pop
  - push
- Member Variable
  - pntr

14.2.4.2 Definition the COBOL class

To ease understanding it is recommended that you use equivalent names in the COBOL class as are used in the C++ class. If names conflict with COBOL reserved words or use non-COBOL characters, alternative names should be found.

The class elements to be defined describes below.

Class

The class name will be used in COBOL to reference the class. STACK is used in this example.

Property

Define as a property that corresponds to a member variable in C++.

In the getting and setting of a property, define the methods of calling interface programs of the member variable reference/set in C++ side outside the program.

In this example, PNTR is defined as a property and as a pointer storage area to maintain objects in the C++ side. In this example, the pointer data is defined by the name of CPP-OBJ-POINTER.

Factory Methods

You need to define the NEW factory method to:

- Create a new object of the COBOL class.
- Call the C++ object creation interface program and acquire the object pointer.
- Store the pointer in the pointer storage area (CPP-OBJ-POINTER).

Object Methods

The method of the same name as the definition of the C++ side is defined as the object method.

Create methods to call the corresponding member function interface programs.

In this example, POP and PUSH are defined as object methods.

Also, create a method DELETE-OBJ to call an object-deleting interface program.

DELETE-OBJ calls the interface program for deleting objects.

14.2.4.3 Define the C++ Interface Programs

The C++ interface programs that you create are described below:

- Object creation interface program.

  The object creation interface program is called by the COBOL NEW method. It creates the C++ object and returns the object pointer.

- Member function access interface program.

  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.

- Member variable reference/set interface program.

  Member variable reference/set interface program is called from the method of property reference/set in COBOL. The argument is a value that is set as object for reference and object for setting. Instance data reference/set program is defined for each member variable.

- Object deletion interface program.

  The object deletion interface program is called by the COBOL DELETE-OBJ method. It deletes the object created in C++.

"Table 14.2 Summary of Interface Programs in Example" summarizes the interface programs created for accessing the example C++ class:

Table 14.2 Summary of Interface Programs in Example

	COBOL class definition	C++ interface program	C++ class definition
Class	STACK	----	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	GET for PNTR	CPP_STACK_GET_PNTR	pntr(member variable)
	SET for PNTR	CPP_STACK_SET_PNTR	

14.2.5 Using the C++ Objects

The code below shows how the C++ object class would be handled from a COBOL program.

Example of use the C++ objects

```
WORKING-STORAGE SECTION.
 01  STACKOBJ USAGE OBJECT REFERENCE STACK.
 01  POP-VALUE PIC S9(9) COMP-5.
PROCEDURE DIVISION.
    INVOKE STACK "NEW" RETURNING 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.
```

14.2.6 The Sample Program

The code samples below show the COBOL interface class in detail and an outline of the C++ code:

COBOL class definition for C++ collaboration

```
*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
*>  DEFINE CLASS STACK
*>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
 IDENTIFICATION DIVISION.
 CLASS-ID.   STACK INHERITS FJBASE.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 REPOSITORY.
     CLASS FJBASE.
*>
*>  FACTORY DEFINITION
*>    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.
```

C++ interface program 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) {
delete *stk;
}
```

14.3 Making Persistent Objects

This section explains how to make persistent objects.

14.3.1 Meaning of Persistent Objects

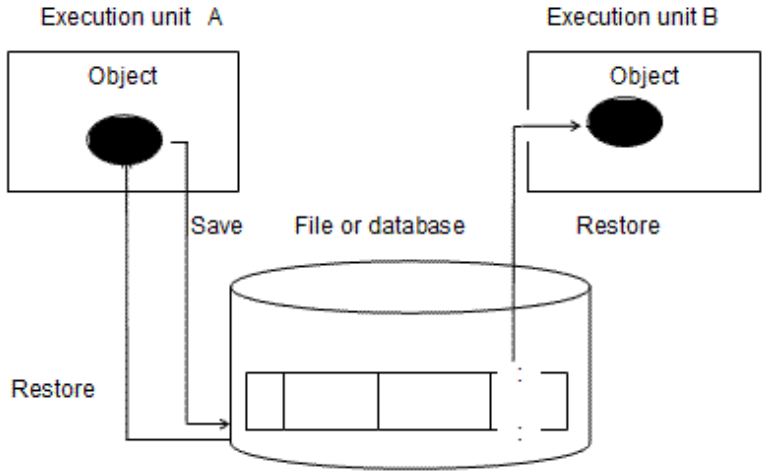
Object-Oriented COBOL objects cease to exist when the application terminates.

For software objects to model real-life objects, they need to exist for the same length of time as real-life objects. In Object-Oriented programming, long-lived objects are called "persistent objects".

14.3.2 Overview

Object-Oriented COBOL does not have persistency built into it, however, you can make persistent objects 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:

Figure 14.4 Flow of persistent object



Add correct identifiers to objects and store/restore according to the identifiers.

Information

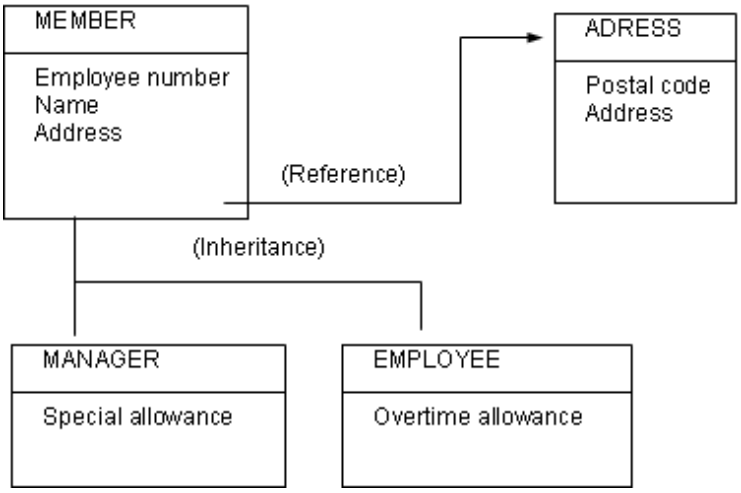
It is actually the object data that is saved and restored, but this section generally describes how to save and restore objects.

This section explains how to make objects persistent using index files.

14.3.3 The Sample Class Structure

The class structure used by explaining this section is as following figure. Only a method and data necessary for the explanation in this figure are shown, and, other method and data are omitted.

Figure 14.5 Sample Class Structure



Explanation of diagram

- 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.

14.3.4 Correspondence of Index Files and Objects

This section explains the correspondence of index files and objects.

14.3.4.1 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.

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.

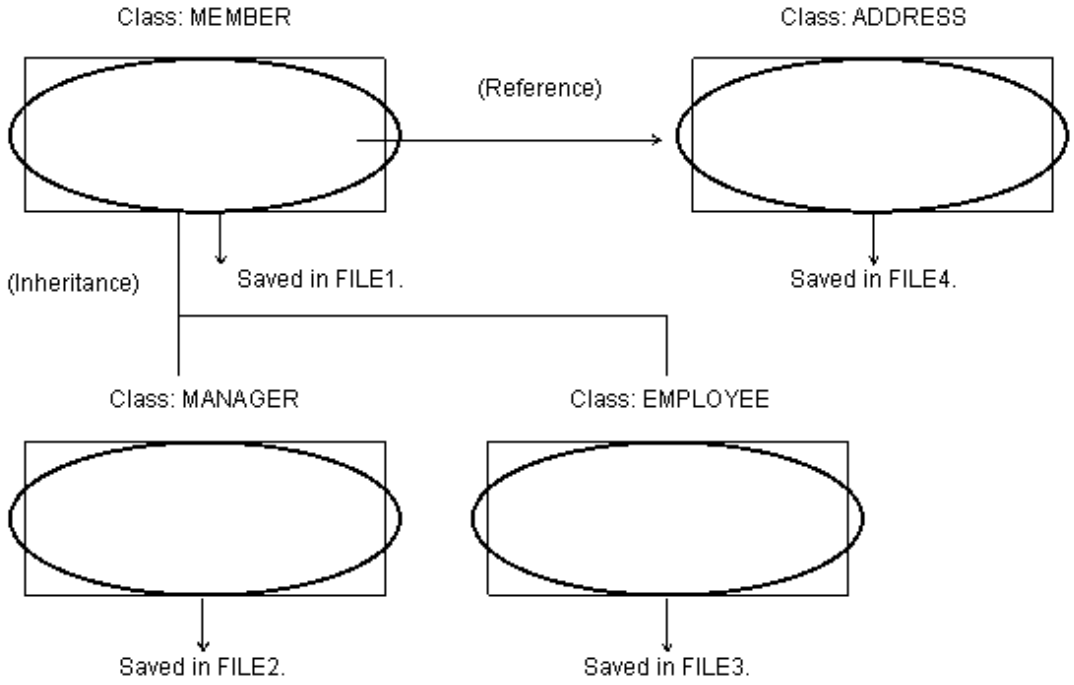
In this model, data that is unique to the MANAGER class is saved to FILE2, and data unique to the MEMBER class is saved to FILE1.

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.

Because there is one file for each class, changing the data structures in a class only affects one file. File maintenance is therefore straightforward.

However, because there are many files involved in saving the structure, managing the files can be complex.

Figure 14.6 One file for each class



One File For Each Elementary Class and its Parents

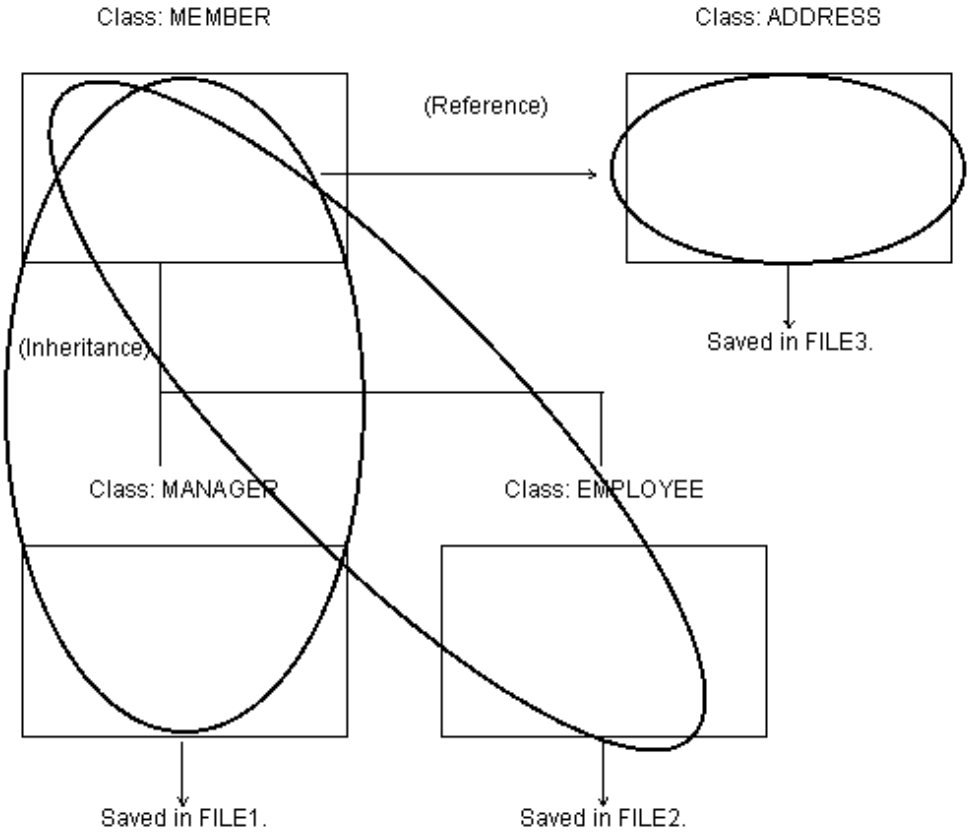
Independent objects as Member class do not exist when the Member class is purely defined as an abstraction class. In this case, it is recommended to save object data of Member class and that of Manager class in the same file by the saving Manager class. This makes processing easier than the method of dividing into several save files by each class. The object data of the Member class and object data of the Employee class can be similarly saved in the same file.

Refer to "Figure 14.7 The method of saving including object data of parent class in one file".

To restore objects of the Manager class, read object data from FILE1 and generate the Manager objects and return them. However, whether the objects are Manager or Employee is information that the class has and is known only after the objects are restored. For instance, it is not possible to decide from which file out of FILE1 and FILE2 should be read for object data with the information on the employee number alone.

This method is effective when there is only one class to be saved. However, when two or more classes inherit a parent class as discussed in "Figure 14.7 The method of saving including object data of parent class in one file" the processing might fail. Also, the files of the class in which the class definition change is in inheritance relationship should be changed.

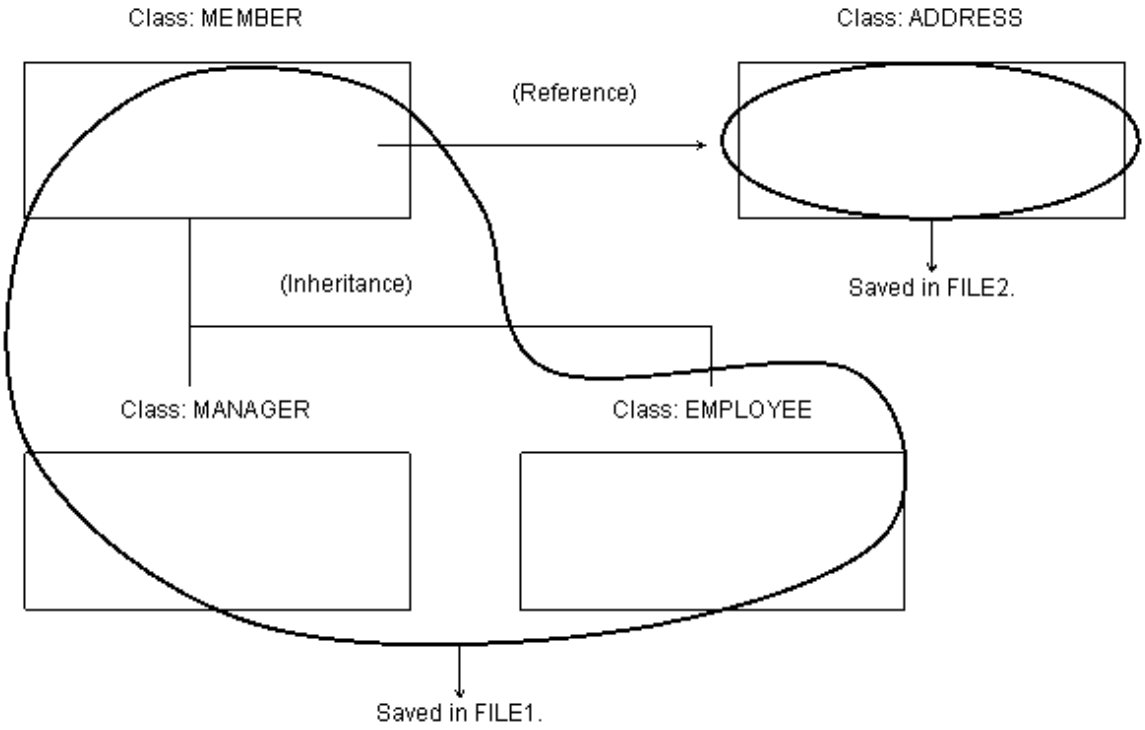
Figure 14.7 The method of saving including object data of parent class in one file



The Method of Saving Classes with the Same Parent Class in One File

It is possible reduce flaws by storing classes that inherit the same parent class in a single file, rather than separate files. Refer to "Figure 14.8 The method of saving classes with the same parent class in one file". With this method, it is possible to restore appropriate objects of Manager and Employee out of the employee numbers.

Figure 14.8 The method of saving classes with the same parent class in one file



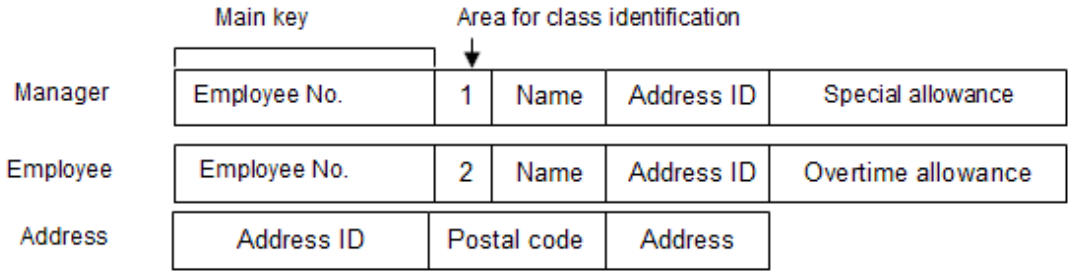
Which of the above-mentioned 3 methods is most suitable depends on the types of the application. This example discusses the suitability according to "Figure 14.8 The method of saving classes with the same parent class in one file".

14.3.4.2 Defining Index Files

Object ID should be unique to objects. In this example, assume the employee numbers are object ID in Manager and Employee that inherit Member and the class and used as a main key of Index file. Also, the area to distinguish whether it is Manager or Employee in the record in the Index files is prepared. Assume Manager is 1 and Employee is 2 in "Figure 14.9 Records in Index files".

Object ID to distinguish objects of Address class should be added. In "Figure 14.9 Records in Index files", the employee number in the address is used as an identifier because the objects of the Address class is referred only as object data of the Member class object.

Figure 14.9 Records in Index files



Address ID is an identifier to identify address objects uniquely.

14.3.5 Saving and Restoring Objects

This section explains saving and restoring objects.

14.3.5.1 Indexed File Handling Class

The method of storing/restoring objects is referred to as Save and Retrieve respectively.

There is a method of writing/reading records by opening/closing index files in the Save/Retrieve too. However, it is not an efficient method because the files are opened/closed at each Save/Retrieve. In this example, create a class that handles files for storing/restoring and open them when the program is started and closed them when the program is terminated.

Define the following methods for the methods of objects of the index file execution class:

- Retrieve:
  Receives an object ID as an argument, reads the file to retrieve the data, and returns the data.
- 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.

The index file operation class defines only the number of files used for storage.

14.3.5.2 Adding Methods of Class to be Stored

Add the following methods to the classes to be stored.

Factory Method

- Retrieve

  The Retrieve method receives the object ID as an argument and restores the corresponding object and returns it. It actually calls the Retrieve method of the corresponding index file operation class. The Retrieve method need not be defined in each class. It is sufficient if the Retrieve method is defined in the parent class.

Object Method

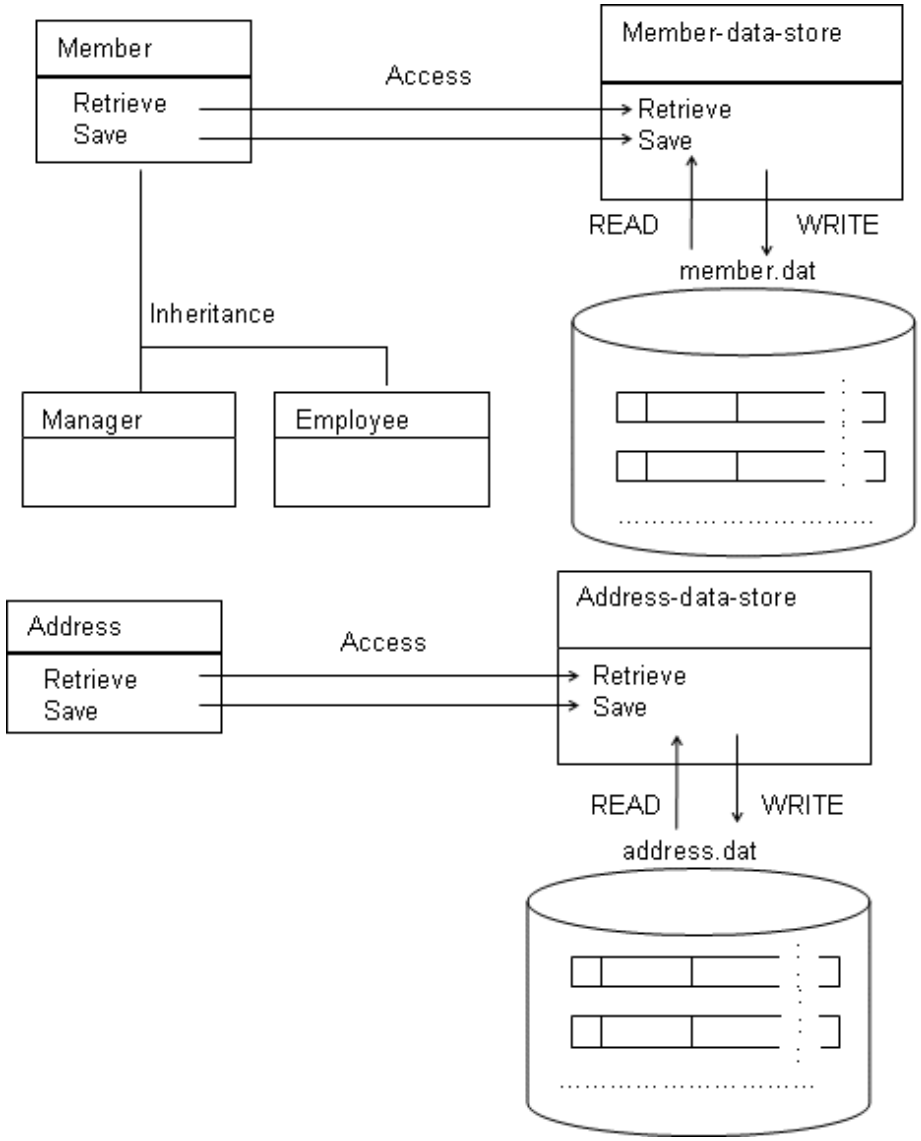
- Save
  The Save method is a method of saving objects. It actually calls the Save method of the corresponding index file operation class with itself as an argument. The Save method need not be defined in each class. It is sufficient if the Save method is defined in the parent class.

The class, index file name, and index file operation class to be saved class are shown in "Table 14.3 Saved class/indexed file/indexed file operating class" below based on the example previously discussed in this chapter. Also, the call relationship of methods and index files is shown in "Figure 14.10 Relationships between persistent classes and supporting file handling classes".

Table 14.3 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	----	address.dat	Address-data-store

Figure 14.10 Relationships between persistent classes and supporting file handling classes



14.3.6 Steps of the Process

The steps of the process of saving/restoring data are explained according to "Figure 14.10 Relationships between persistent classes and supporting file handling classes".

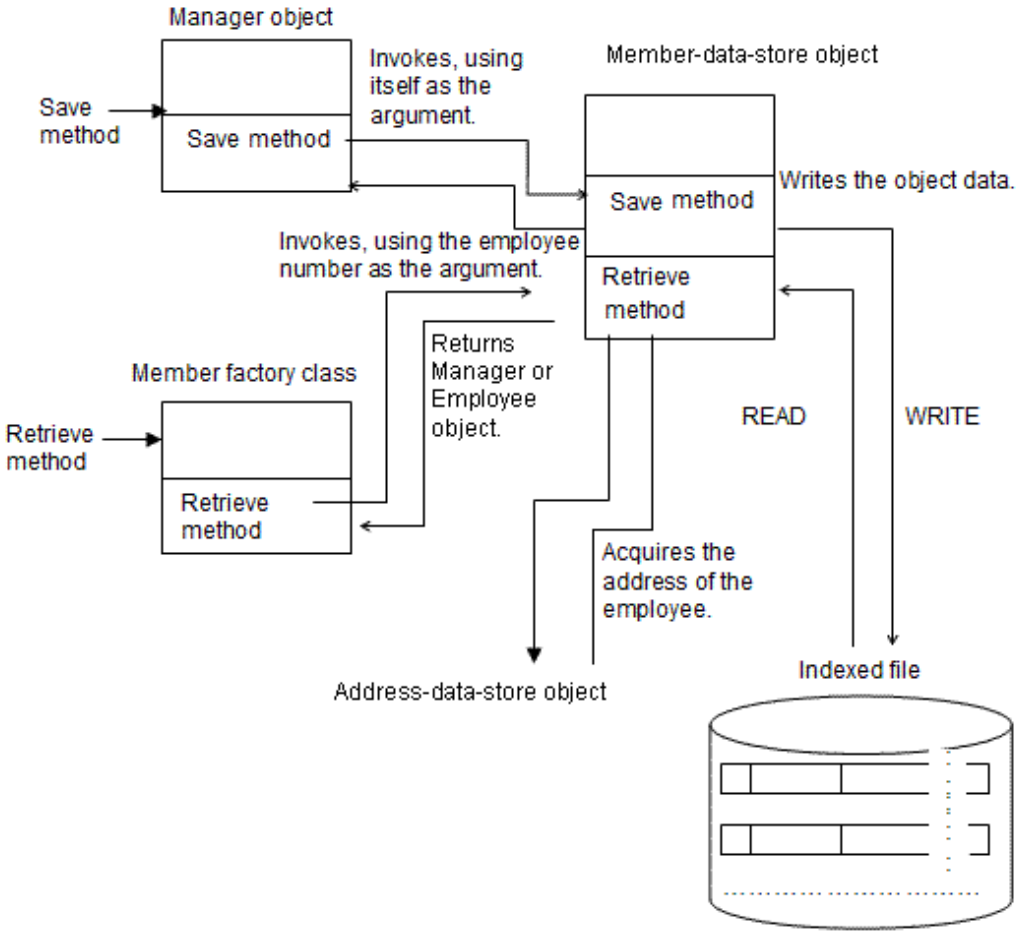
Save

When the Save method is called for the Manager object, the Save method makes itself an argument. Then the Save method of the index file operation object (Member-data-store object) for Member is called. The Save method of the Member-data-store object examines the class of the object of the argument (Manager class in this case). Data is then posted on the record for the Manager class and written in the index file.

Restore

The Retrieve method calls the Retrieve method for corresponding index file operation objects (Member-data-store object) making the employee numbers to be arguments. The Retrieve method of the Member-data-store objects reads the record from the index files with the employee number as a main key and examines the sub-key. Then it returns the object of the Manager class or the Employee class. The Retrieve method also uses the employee numbers as arguments. The Retrieve method of the Address-data-store object is called. It then retrieves the employees' address (Address) objects. The retrieved Address objects are posted in the address of the object data (Information about restoring the Address object is omitted in "Figure 14.11 Flow of processing of save/restore").

Figure 14.11 Flow of processing of save/restore



14.4 Programming Using the ANY LENGTH Clause

This section explains to create applications using ANY LENGTH clause for data items.

14.4.1 Class that Handles Character Strings

When a class that handles various lengths of character strings is created using the object-oriented function, the maximum length of character strings to be handled must be determined, 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 of the object-oriented function. This means that even if the calling side wants to pass a character string of different length, 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 at the first time,

1. Change the maximum lengths of character strings in the class where the method to be called is defined, and those in the programs and classes referencing the class, to the same length.
2. Then you must recompile all of the changed classes or programs.

If the maximum length is decided 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, and 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 20 alphanumeric characters, and the password as 8 alphanumeric characters.

Before character strings are passed:

```
 PROGRAM-ID. INFORMATION.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  REPOSITORY.
      CLASS  CONFIRM-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  CONFIRM-CLASS.
 PROCEDURE DIVISION.
     INVOKE  CONFIRM-CLASS "NEW" RETURNING  AUTHENTICATED-OBJECT.
     DISPLAY "ENTER THE NAME AND PASSWORD".
     ACCEPT  NAME-STRING.
     ACCEPT  PASSWORD-DATA.
     INVOKE  AUTHENTICATED-OBJECT "CONFIRM-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.
```

General-purpose class that handles passed character strings:

```
  CLASS-ID.   CONFIRM-CLASS INHERITS FJBASE.
*>  :
  OBJECT.
  PROCEDURE DIVISION.
  METHOD-ID.  CONFIRM-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 SMITH" ALSO "A1"
       WHEN   "KEN WILLIAMS" ALSO "XXXXYYY2"
         MOVE  "OK" TO  AUTHENTICATION-RESULT
       WHEN OTHER
         MOVE  "NG" TO  AUTHENTICATION-RESULT
       END-EVALUATE.
  END METHOD  CONFIRM-METHOD.
  END OBJECT.
  END CLASS   CONFIRM-CLASS.
```

If another process using the CONFIRM-CLASS is generated under the above conditions, and the maximum length of the password is to be changed, the CONFIRM-CLASS must first be modified, thereby triggering the need to modify the information change program. Also, after changing the interface, the class inheriting the CONFIRM-CLASS must be recompiled.

14.4.2 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.   CONFIRM-CLASS INHERITS FJBASE.
*>        :
 OBJECT.
 PROCEDURE DIVISION.
 METHOD-ID.  CONFIRM-METHOD.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LENGTH_OF_NAME     PIC 9(4) COMP-5.
 01  LENGTH_OF_PASSWORD PIC 9(4) COMP-5.
 LINKAGE SECTION.
 01  NAME           PIC X  ANY LENGTH.        *>-+ When CONFIRM-METHOD is passed,
 01  PASSWORD-DATA  PIC X  ANY LENGTH.        *> | the length of the corresponding item
 01  AUTHENTICATION-RESULT PIC X  ANY LENGTH. *>-+ on the invoking side is determined.
 PROCEDURE DIVISION USING  NAME  PASSWORD-DATA
                   RETURNING  AUTHENTICATION-RESULT.
     COMPUTE LENGTH_OF_NAME       = FUNCTION LENGTH(NAME).
     COMPUTE LENGTH_OF_PASSWORD   = FUNCTION LENG(PASSWORD-DATA).
*>    :
  END METHOD  CONFIRM-METHOD.
  END OBJECT.
  END CLASS   CONFIRM-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.

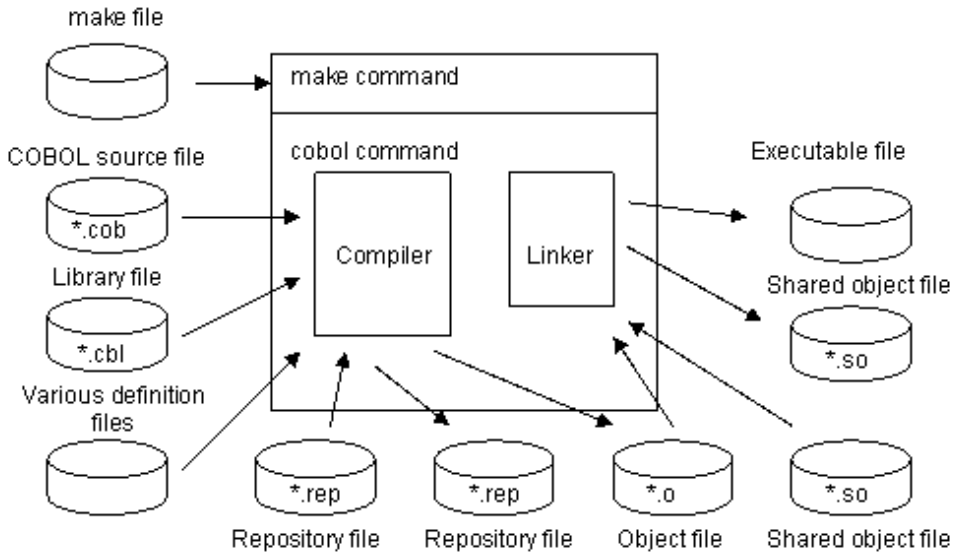
Chapter 15 Developing and Executing Object-Oriented Programs

This chapter explains how to develop and execute Object-Oriented programs.

15.1 Resources Used by the Object-Oriented Programming

This chapter explains the resources to develop Object-Oriented COBOL Applications. In object-oriented programming, repository files are added to conventional development resources.

The relationship between resources of object-oriented programming is shown below:



Files used in Object-Oriented programming are described below:

Table 15.1 Files used in Object-Oriented programming

File type	File contents description	File-name format	Input/output	Operation or creation condition
Repository file	Class-related Information for inheritance and conformance check	class-name.rep	Input	Used when source files specified the REPOSITORY paragraph is compiled.
			Output	Output created when class definition COBOL source files are compiled.

15.2 Development Steps

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 make command because dependent relationships of the resource may often become complex when compiling and linking the classes. For more information about the make command, refer to "Appendix L Using the make 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 object-oriented 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

As explained in "12.3 Concepts of Object-Oriented Programming", Object-oriented 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.

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.

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 methods that execute the processing that suits the purpose, you can 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, and the user knows the name of that method, the user can 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.

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 object-oriented programming, the following definitions are considered to be a unit of one compilation:

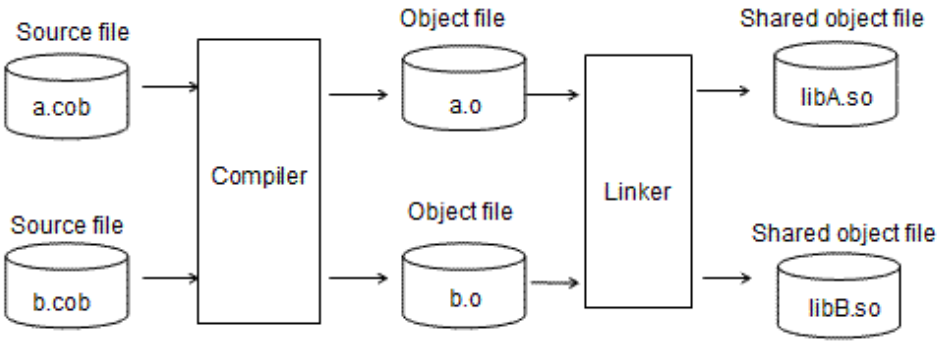
- Class definition.
- Program definition.
- Method definition separated by PROTOTYPE declaration.

Also, a link unit means the unit for which one executable file or shared object file is created by a link processing.

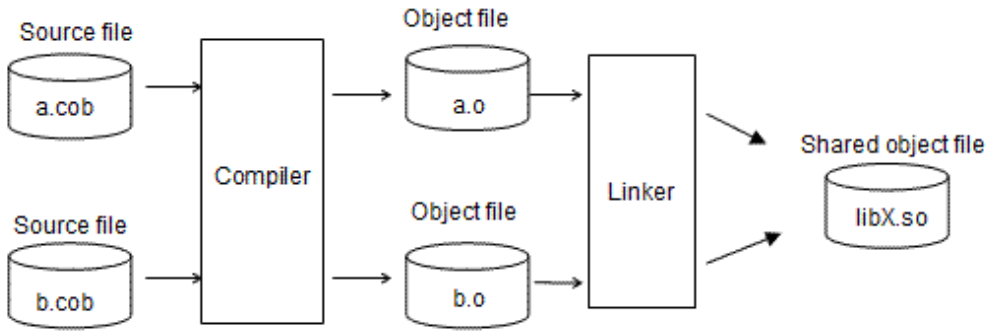
It is also possible that one executable file or shared object file is composed of more than one object file. Therefore, compilation units and link units might not correspond.

The figures below show the relationship between compilation unit and link unit:

When a compilation unit corresponds to a link unit:



When a compilation unit does not correspond to a link unit:



15.5.2 Overview of Program Structure

As explained in "3.2.2 Linkage Type and Program Structure", the following linkages are supported as the linked program structure:

- Static linkage.
- Dynamic linkage.

The following section explains the case where object-oriented programs are linked by a static linkage or a dynamic linkage.

15.5.2.1 Static Linkage

An executable file or shared object file is composed of more than one object file in a static linkage.

Also, when an object file of a general-purpose program or class is created, it should be embedded in all executable files or shared object files that use it.

Therefore, a static linkage is used in the following case where the link relationship is specified in a few patterns:

- Class definition that contains methods declaring PROTOTYPE and method definition separated by it.

15.5.2.2 Dynamic Linkage

The dynamic link structure and dynamic program structure are supported as dynamic linkage.

Dynamic Link Structure

In the dynamic link structure, more than one shared object file created by a certain function unit are linked when an executable file is started.

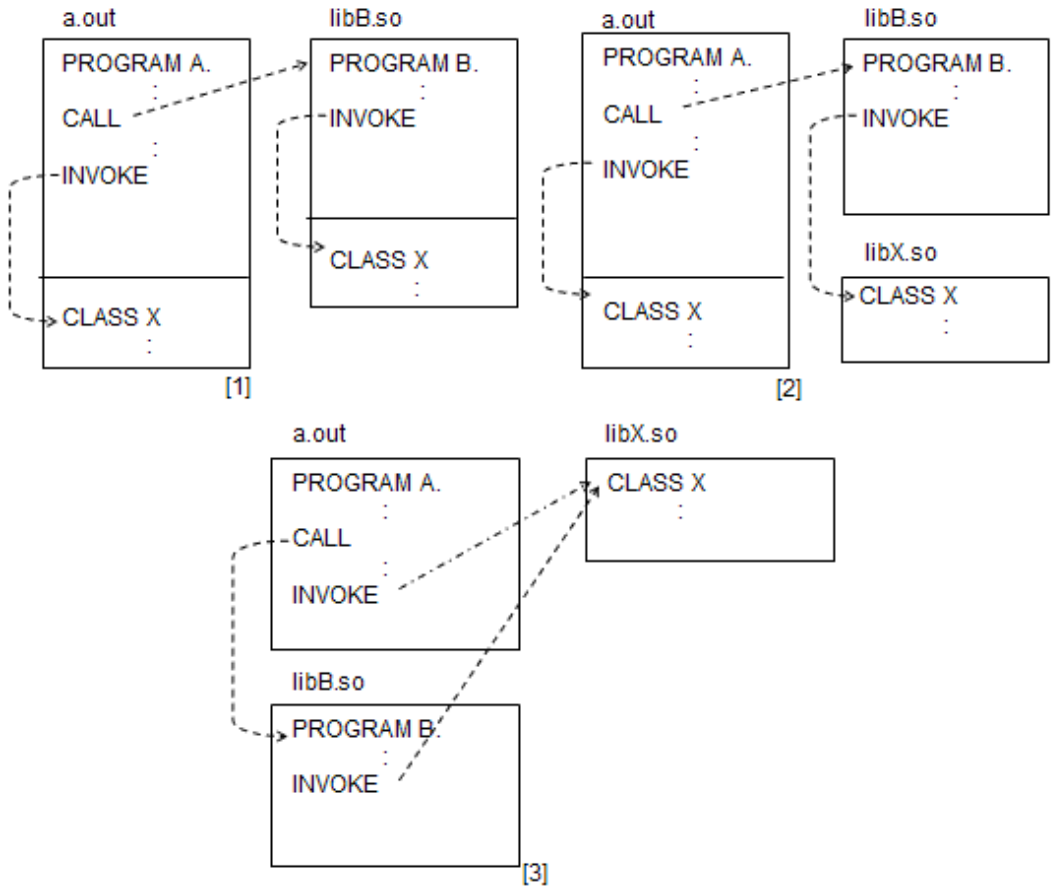
Unlike static linkage, the dynamic link structure only has to link shared object files to be compiled even when a source file is corrected and recompiled. (Other executable files and shared object files are not affected.)

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 two scenarios, [1] and [2], in which two copies of Class X executable code are loaded into the same execution unit. The correct way to structure the application is shown in diagram [3].



The dotted lines indicate the relationship for call/reference.

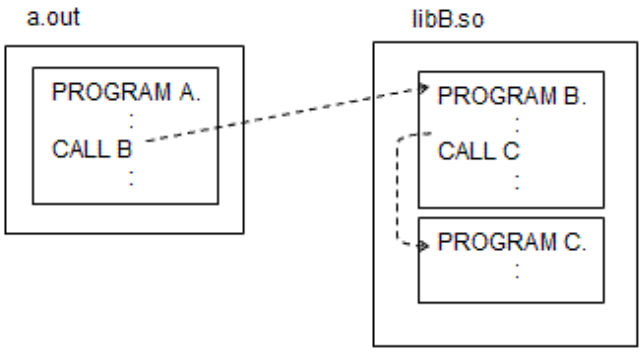
- [1] Class X is statically linked with program A and statically linked with program B.
- [2] Class X is statically linked with program A and invoked dynamically from program B.
- [3] Class X is invoked dynamically from both program A and program B. (Structure could be either dynamic link or dynamic 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:



Dynamic Program Structure

In the dynamic program structure, classes and methods are loaded on a virtual storage by the runtime system of COBOL when they are actually referred or called.

Therefore, the startup of an executable file is quicker compared with a simple structure or the dynamic link structure into which all files are loaded when executable files are started. However, the execution of the statement referring to classes and calls of methods is slower because they are executed through the runtime system of COBOL.

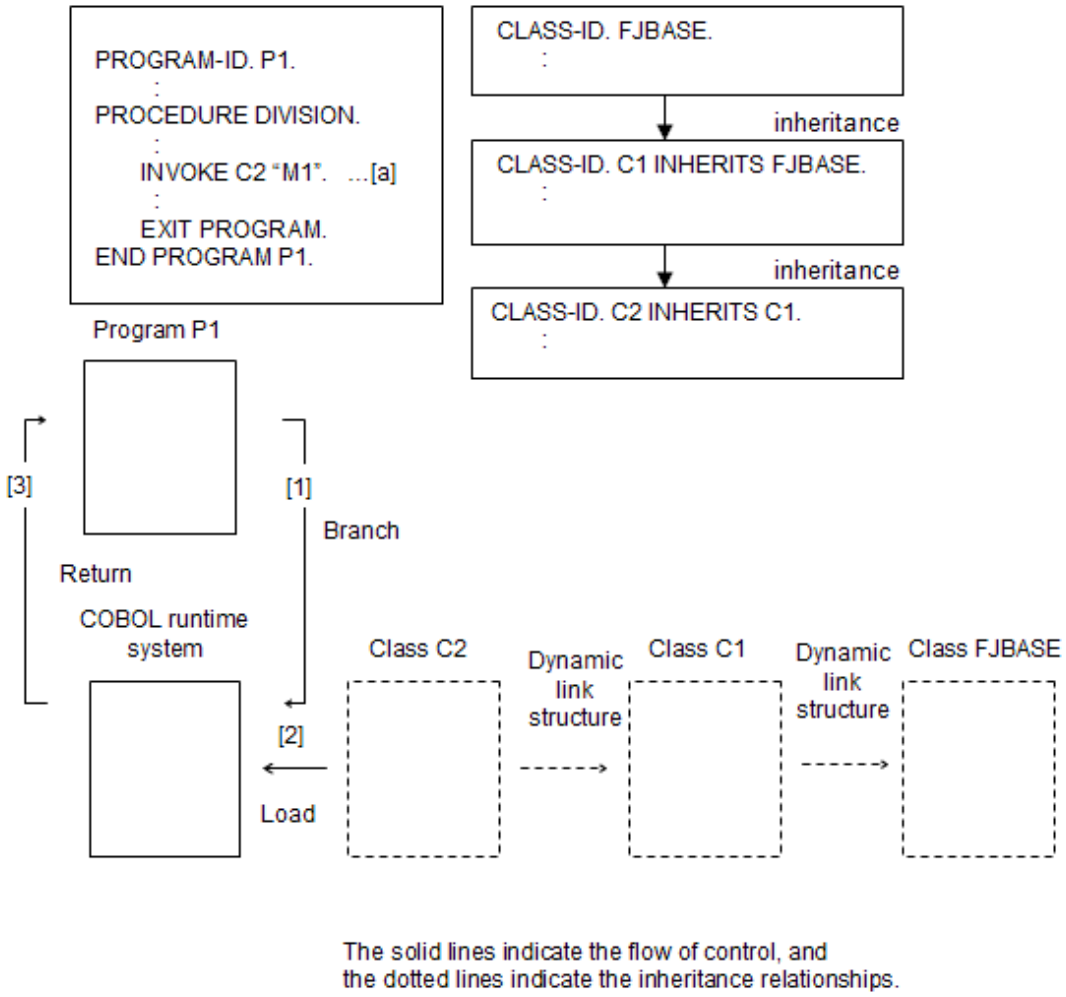
Note

- Class entry information is required for executing dynamic program structure classes. However, the entry information file is unnecessary if the shared object file name of the class is "libclass-name.so". For this reason, it is recommended that you make create a shared object file for each class that is called using the dynamic program structure, and that you make the file name "libclass-name.so".
- In the conventional program that is not object-oriented program, programs loaded by a dynamic program structure can be deleted from memory with a CANCEL statement. However, no function equivalent to the CANCEL statement exists in object -oriented programs. Therefore, you cannot take advantage of this.
- For details how to use this with the program dynamic program structure, refer to "8.1.2.3 Cautions".

Dynamic Program Structure of Class

If a class is a dynamic program structure, the loading of the class used by an application onto the virtual storage is executed by the runtime system of COBOL when the class is referred. Then the class that directly or indirectly inherits to the class is loaded at the same time. This is because the class that is directly or indirectly inherited by the class is a dynamic link structure.

In the following programs and if class C2 is the dynamic program structures, class C2 is loaded onto the virtual storage when the INVOKE statement of [a] is executed. Class C1 directly inherited by class C2 and class FJBASE indirectly inherited by class C2 are loaded then.



Explanation of figure

([1]-[3] shows the processing order.

- [1] Class C2 is referred by executing the INVOKE statement, which calls COBOL runtime system.
- [2] COBOL runtime system loads class C2. At this time, class C1 and FJBASE inherited by class C2 are loaded by the system.
- [3] Returns to program P1.

Note

- When the class is a dynamic program structure, the call of the method becomes a dynamic program structure by the PROTOTYPE declaration as well.
- File names must be specified in the following format for the method used by the dynamic program structure ("Separated method"):

  ```
  "lib-class-name_method-name.so"
  ```

  Specify the method name of the class for which the prototype is defined in the property method as follows when the property method is a dynamic program structure.

  - For GET method

    ```
    "lib-class-name__GET_property-name.so" (*)
    ```

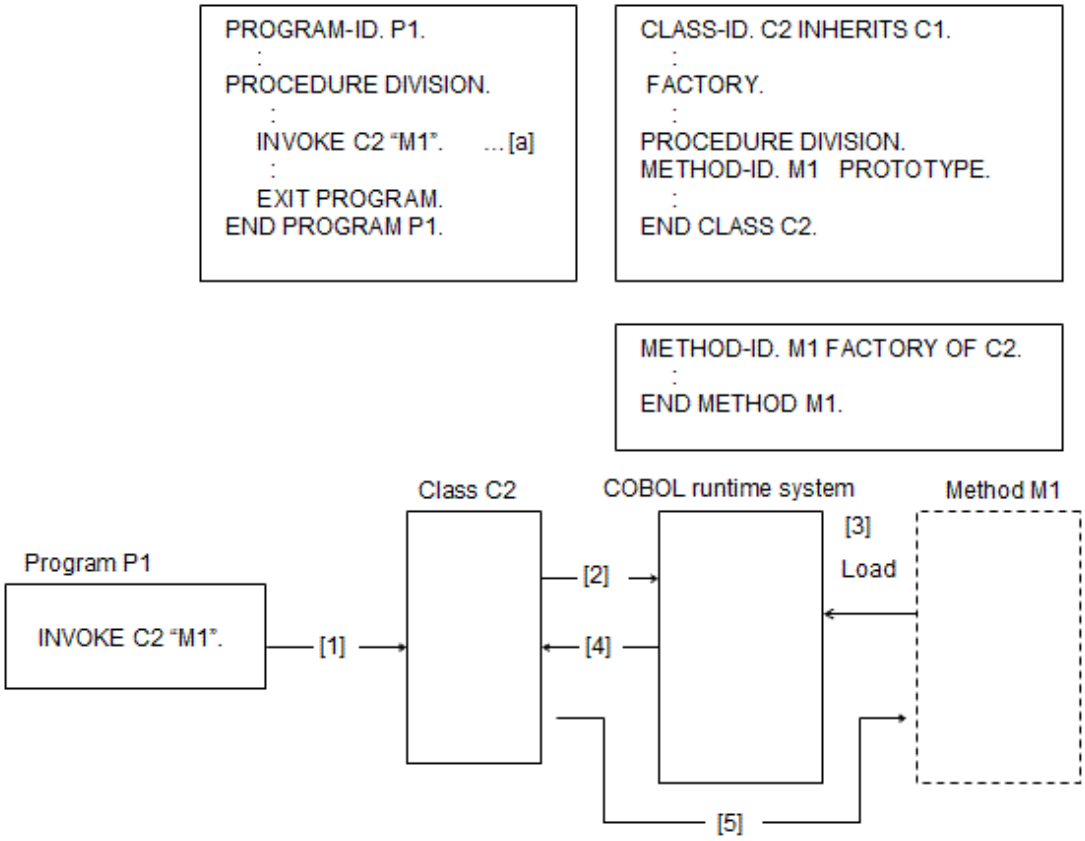
- For SET-method

  ```
  "lib-class-name__SET_property-name.so" (*)
  ```

* : There must be 2 single-byte underscores (_) between the class name and GET or SET.

Dynamic Program Structure of Methods

The methods separated by a PROTOTYPE declaration can be created into dynamic program structures. By doing so, the loading of the methods used in the application on the virtual storage will be executed by the runtime system of COBOL when the method is invoked. When the INVOKE statement of [a] is executed, method M1 is loaded on the virtual storage when they are the following programs and method M1 is the dynamic program structures:



Explanation of figure

[1]-[5] shows the processing order.

- [1] Class C2 is used by invoking method M1.
- [2] COBOL runtime system is called.
- [3] COBOL runtime system loads method M1.
- [4] Returns to class C2.
- [5] Method M1 is invoked.

15.6 Compilation

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.

- Compilation processing in dynamic program structures.

For information about general compilation processing, refer to "Chapter 3 Compiling and Linking Programs".

15.6.1 Repository File and Compilation Procedure

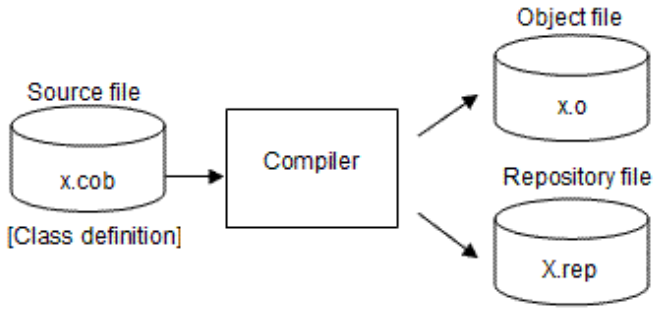
This section explains repository files first.

Overview

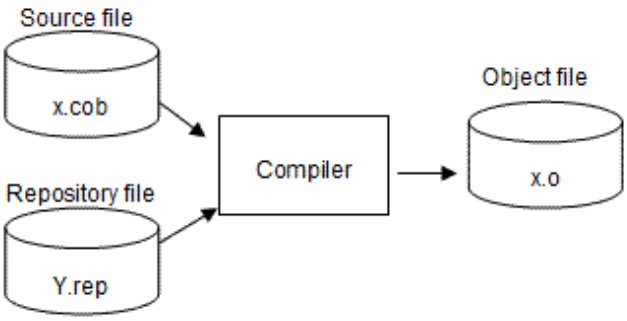
Repository file stores information on classes generated when the class definition is compiled. Repository files are used to notify compilers of information on the classes to be reused when compiling.

The file name of repository file is "Class-name (externalized name).rep".

Outputting Repository Files



Inputting Repository Files



Compilers input to only repository files that the 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.          *>   [a]
 ENVIRONMENT DIVISION.
  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  Address-Ref      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.       *>[a]
 ENVIRONMENT DIVISION.
  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)

Compilation Procedure

When a 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 to. A compilation error may occur if the necessary repository file does not exist.

When the class definition is recompiled, the repository file is updated also. In that case, the source files that have input the updated repository files in the compilation should be recompiled.

Thus, there is a restriction in the compilation order depending on the relationships of the input of repository files.

The file "member.cob" is described as follows:

```
    :
CLASS-ID. Member-class INHERITS AllMember-class.
```

```
    :
 REPOSITORY.
    CLASS AllMember-class.
    :
```

You can tell that the repository file of the AllMember-class class is needed in the compilation from the contents of this program.

The file "allmem.cob" is described as follows:

```
    :
CLASS-ID. AllMember-class INHERITS FJBASE.
    :
 REPOSITORY.
    CLASS FJBASE
    CLASS Address-class.
    :
```

You can tell that the repository file of the FJBASE class and the Address-class class is needed in the compilation from the description of this program.

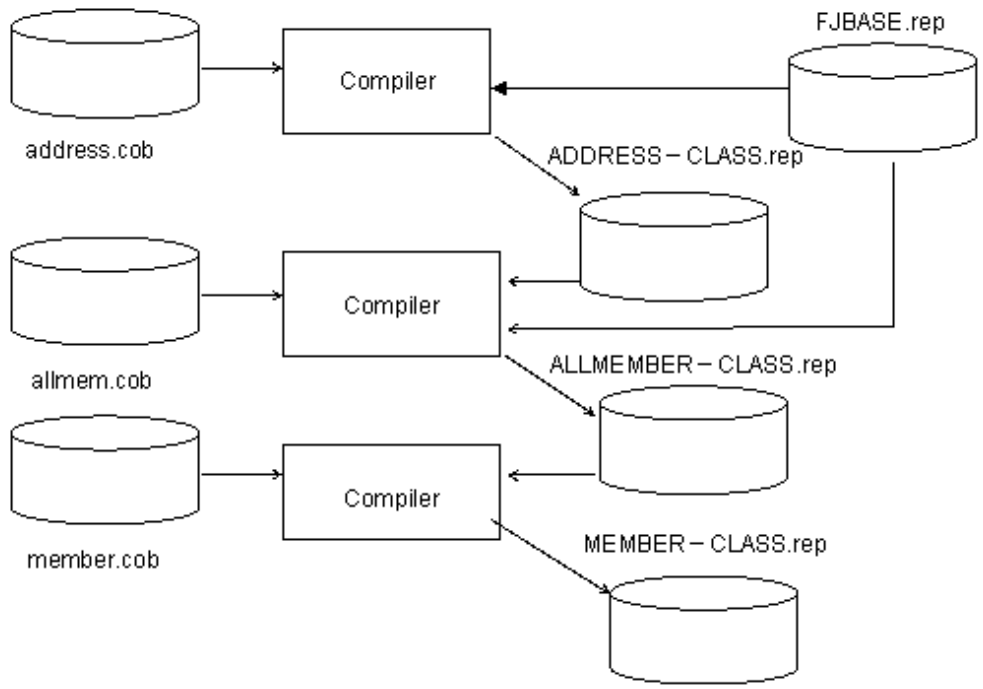
The file "address.cob" is described as follows:

```
    :
CLASS-ID. Address-class INHERITS FJBASE.
    :
 REPOSITORY.
    CLASS FJBASE.
    :
```

You can tell that the repository file of the FJBASE class is needed in the compilation from the description of this program.

From the previous discussions above, the relationships between the resources can be summarized as follows:

Figure 15.1 The compilation order



The compilation processing should be executed in the following order as understood from "Figure 15.1 The compilation order":

1. address.cob
2. allmem.cob

3. member.cob

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" in "Figure 15.1 The compilation order" is "MEMBER-CLASS.rep".

Target Repository File Directory

The directory in which target repository file is generated when the class definition is compiled is decided by specifying the compilation command option -dr as shown in the table below.

For more information, refer to "3.3.1.5 -dr (Specification of input/output destination directory of the repository file".

-dr option	Directory name
Valid	Directory specified by the -dr option.
Invalid	Directory containing the COBOL source file.

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" in "Figure 15.1 The compilation order" is "ALLMEMBER-CLASS.rep".

Note

The dependent repository file is not needed for the FJBASE class. For details on the FJBASE class, see "13.3.2 FJBASE Class".

Dependent Repository File Search Paths

When the compiler searches a repository, it uses the pathname specified in the following table:

-R option	-dr option	Order of Search Paths
Valid	Valid	(1) Path specified by the -R option. (2) Path specified by the -dr option. (3) Current path.
	Invalid	(1) Path specified by the -R option. (2) Current path.
Invalid	Valid	(1) Path specified by the -R option. (2) Current path.
	Invalid	(1) Current path.

For more information, refer to "3.3.1.5 -dr (Specification of input/output destination directory of the repository file" and "3.3.1.16 -R (Specification of repository file input destination directory)".

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 "3.3 cobol Command".

```
$  cobol -c -R /home/cobol -dr /home/cobol allmem.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

Input

allmem.cob (Source file)

ADDRESS-CLASS.rep (Repository file: Stored in /home/cobol)

MEMBERMASTER-CLASS.rep (Repository file: Stored in /home/cobol)

Output

allmem.o (Object file)

ALLMEMBER-CLASS.rep (Repository file: Stored in /home/cobol)

Option

-c (Specification of executing only compilation)

-R (Directory of input destination of repository file)

-dr (Directory of input and output destination of repository file)

Compiling Cross Reference Classes

Techniques are needed to create dependent repository files to compile cross reference classes. For more information about the compilation of the cross reference classes, refer to "13.7.8.2 Compiling Cross Reference Classes".

15.6.2 Compilation Processing in Dynamic Program Structures

This section explains the compiler option and notes needed for compilation.

When You Make a Class into a Dynamic Program Structure

To make classes dynamic program structures, specify the compiler option DLOAD when source programs referring to the classes are compiled. The classes referred from this source program become dynamic program structures by this specification.

For more information, refer to "A.2.11 DLOAD (program structure specification)".

However, if the class is a dynamic link structure in other source programs, the class is loaded by the system when an executable file is started. Therefore, standardize the structures of the program in the run unit of COBOL.

Making a Method a Dynamic Program Structure

To make a method a dynamic program structure, define methods to be dynamic program structure into the prototype and specify the compiler option DLOAD when the class that contains the prototype definition of the method is compiled. Thus, the program structure of the separated method is decided according to how the classes are compiled.

To make a property method into a dynamic program structure, users should define the property method into the prototype and specify the compiler option DLOAD when the class that contains the prototype definition of the method is compiled.

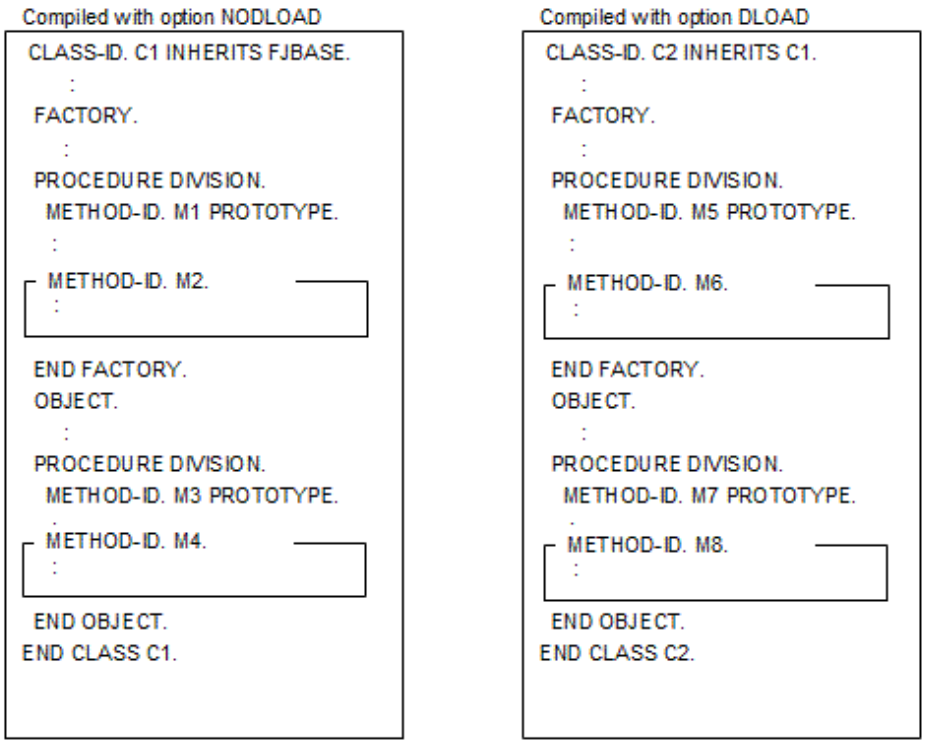
For more information, refer to "A.2.11 DLOAD (program structure specification)".

Example

Assume that there are class C1 that has been compiled in the compiler option NODLOAD and class C2 that has been compiled in the compiler option DLOAD and they are in the inheritance relationship as follows.

In this case, method M5 and method M7 that have been defined into the prototype out of the methods defined in class C2 are dynamic program structures. Even when class C2 is compiled in the compiler option DLOAD, it does not become a dynamic program structure because the prototype has not been defined in method M6 and method M8.

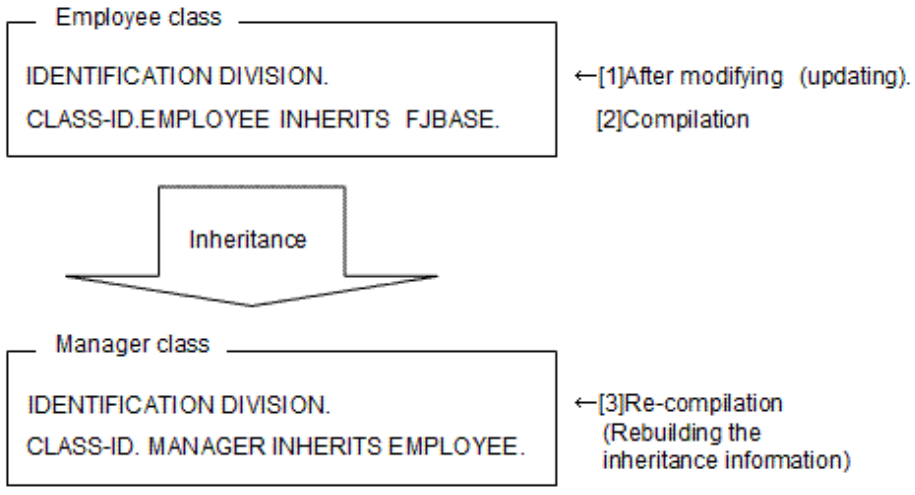
The method defined in class C1 does not become a dynamic program structure because class C1 is compiled in the compiler option NODLOAD.



Influence of Repository File Update

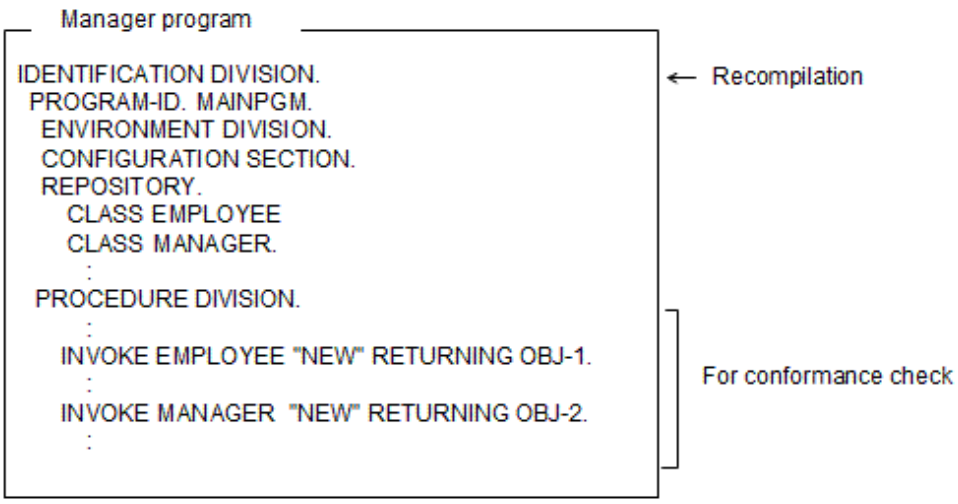
Note the following points when parent classes or classes to be called are corrected after generating object programs.

Compilers achieve inheritance and conformance check only from information stored in repository files. Therefore, when repository files are updated, restructuring of inheritance information and re-executing of conformance check is necessary. In other words, when the interface of the parent class is corrected, any child class should be recompiled even if there is no correction. It should be recompiled if there is correction with a change in the interface as well.



The child class (Manager class) of the corrected class (Employee class) needs to be recompiled to restructure inheritance information even if it has not been corrected at~all.

Also, the programs and classes calling the corrected classes need to be recompiled for the conformance check.



Users should execute these recompilations. Therefore, pay much attention when you correct the class definitions once constructed.

15.7 Linking Programs

This section explains the following two points about link process for development of object-oriented programs.

15.7.1 Linkage and Link Procedure

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 object-oriented programming.

"Table 15.2 Link in object-oriented programming" shows the cases where linkage is needed in object-oriented programming:

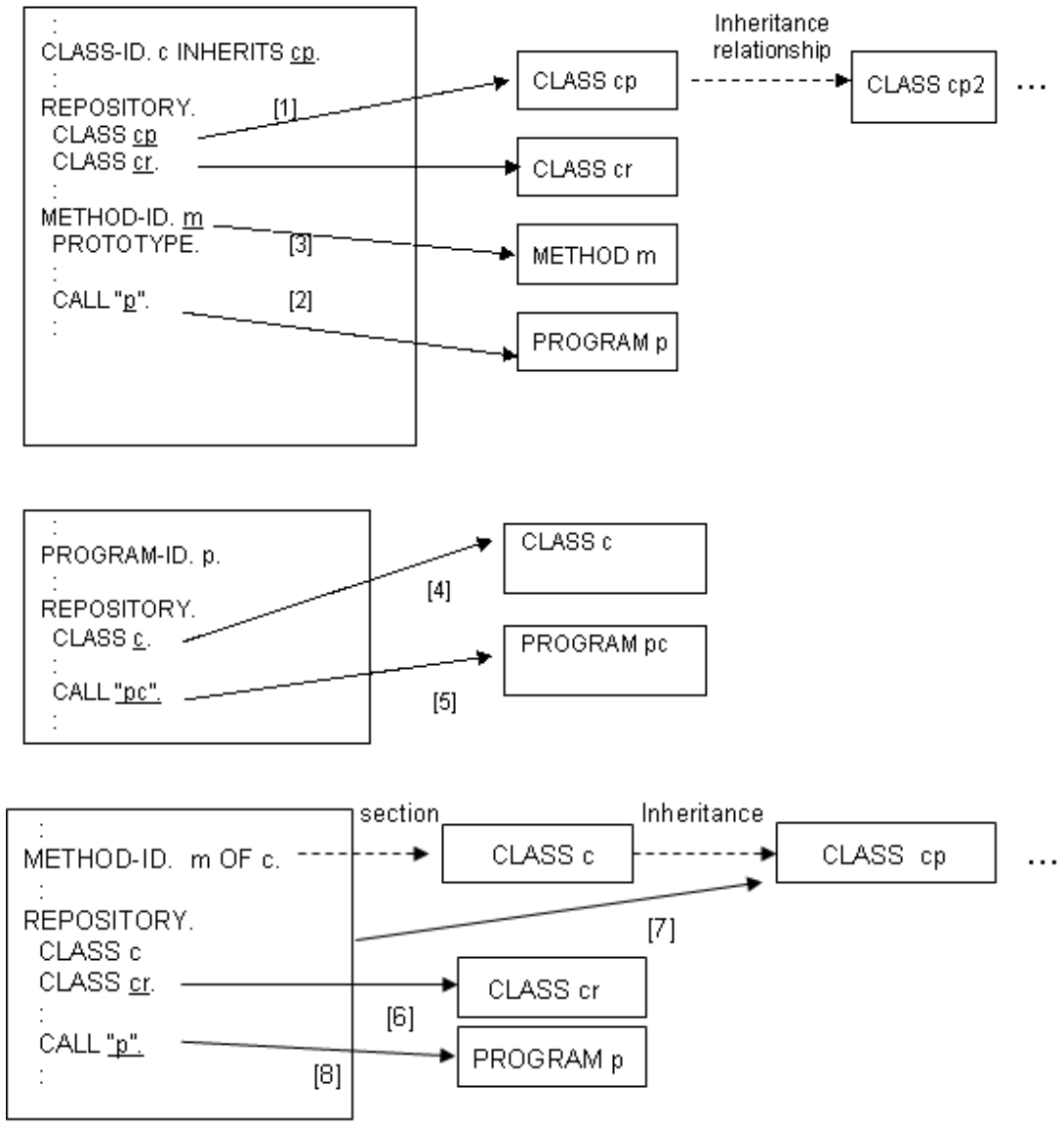
Table 15.2 Link in object-oriented programming

Definition name		Link to		
		Class definition	Program definition	Method definition
Link from	Class definition	[1] Class described in REPOSITORY paragraph.	[2] Program called by CALL statement.	[3] Method separated by PROTOTYPE declaration.
	Program definition	[4] Class described in REPOSITORY paragraph.	[5] Program called by CALL statement.	------
	Method definition	[6] Class described in REPOSITORY paragraph (except for classes including PROTOTYPE declaration of its own method). [7] Parent class including PROTOTYPE declaration of its own method. (*)	[8] Program called by CALL statement.	------

* : This applies only when SUPER has been specified within PROCEDURE DIVISION.

The link in each definition is shown in Figure below.

Figure 15.2 Links in each definition



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, shared object files are needed.

Note

Shared object files that are the parent class in inheritance relationship should be solved with the dynamic link structure.

Procedure of Link

When executable files or shared object files are created using a linker, shared object files to be linked are needed. Therefore, when two or more executable files or shared object files 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>

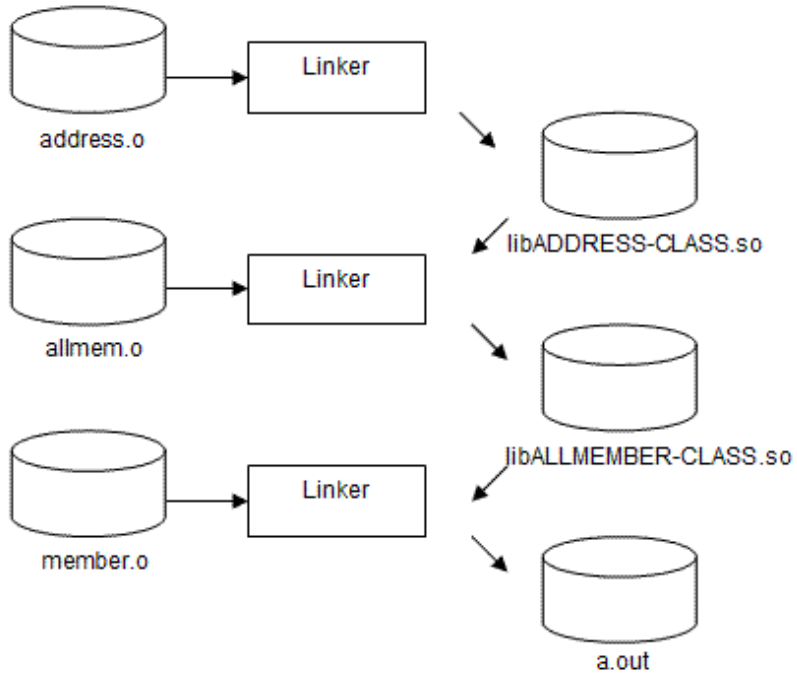
```
     :
CLASS-ID. Address-class ...
     :
```

The following tables shows what shared object files needed for each link procedure and executable file to be generated in the case where the compilation unit and link unit are made to be the same:

Source files or Object files	Executable files or shared object files	Shared object files to be used
member.cob member.o	a.out	libALLMEMBER-CLASS.so
allmem.cob allmem.o	libALLMEMBER-CLASS.so	libADDRESS-CLASS.so
address.cob address.o	libADDRESS-CLASS.so	------

This relationship is shown in Figure below.

Figure 15.3 The order of shared object file and link procedure



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 "Figure 15.3 The order of shared object file and link procedure":

1. address.o
2. allmem.o
3. member.o

Example of Linking by cobol Command

Example of linking by cobol commands is shown below.

```
$  cobol -shared -o libADDRESS-CLASS.so address.o
$  cobol -shared -o libALLMEMBER-CLASS.so -L. -l ADDRESS-CLASS allmem.o
$  cobol -o a.out -L. -l ALLMEMBER-CLASS member.o
```

Input

address.o (Object file)

allmem.o (Object file)

member.o (Object file)

Output

libADDRESS-CLASS.so (Shared object file)

libALLMEMBER-CLASS.so (Shared object file)

a.out (Executable file)

Option

-o(Output destination of executable files or shared object files)

-l(Specification of libraries to be linked)

-L(library-search-path-name)

Information

Executable programs can be generated by using the ld command. For information about how to use the ld command, refer to "Appendix K ld Command".

15.7.2 Link Procedure in Dynamic Program Structures

This section explains shared object files needed in linkage.

To Create Executable Files or Shared Object Files of Programs

Shared object files are unnecessary.

To Create Shared Object Files of the Class

Only shared object files that are directly inherited by any classes that create shared object files are needed.

To Create Shared Object Files of Separated Methods

the procedure division of a separate method that creates a shared object file requires the shared object file of the class inherited directly by the class where the method is defined.

15.7.3 Shared Object File Configuration and File Name

This section explains the shared object files for the class and method, and standard configuration and file names. According to the examples shown below, you can omit the entry information file.

Shared Object File Configuration and File Name of Class

Create the shared object in class units, and make the file name "lib*class-name*.so".

Shared Object File Configuration and File Name of Method

Create the shared object in method units, and make the file name "lib*class-name_method-name*.so". "class-name" shows the class name of the class defined in the method prototype.

15.7.4 Class and Method Entry Information

This section explains the entry information file required to execute the dynamic program structure application.

If the subprogram is dynamic program structure, subprogram entry information is required. If the class is dynamic program structure, class entry information is required. If the method is dynamic program structure, method entry information is required.

Specify class or method entry information file name in the environment variable CBR_ENTRYFILE in the same way as for subprogram entry information.

For details about subprogram entry information, refer to "4.1.3 Subprogram entry information".

Class Entry Information

Class entry information is information for linking a class and the common object file that class is stored in.

The class entry information description format is as follows:

```
[CLASS]                                          ...[1]
class-name=shared-object-file-name               ...[2]
```

Diagram explanation

- [1] Section name that shows the start of the class entry information definition

  The section name is fixed as "CLASS". You can only describe this section once in an entry information file.

- [2] Class entry information

  Specify the class name of the class to be used in "class-name". Specify the absolute or relative path of the shared object file that contains the class in "shared-object-file-name". The file extension of the shared object file must be "so".

  If the shared object file name is "lib*class-name*.so", you can omit the class entry information.

Method Entry Information

Method entry information is information for linking a method and the shared object file that method is stored in.

The method entry information description format is as follows:

```
[class-name.METHOD]                              ...[1]
method-name=shared-object-file-name              ...[2]
```

Diagram explanation

- [1] Section name that shows the start of the method entry information definition

  The section-name is the character string which added the fixed string "METHOD" to the class name in which the method is defined. You can only describe this section once for a class in an entry information file.

- [2] Method entry information

  Specify the method name in the "method-name". Specify the absolute or relative path of the shared object file that contains the method in "shared-object-file-name". The file extension of the shared object file must be "so".

  If the common object file name is "lib*class-name_method-name*.so", you can omit the method entry information.

Information

If the property method is dynamic program structure, you must specify the method name in the method entry information of the class in which the property method is defined as follows:

- GET method

  Specify "_GET_*property-name*". However, if the GET method shared object file name is "lib*class-name*_GET_*property-name*" (*), you can omit the method entry information.

- SET method

  Specify "_SET_*property-name*". However, if the SET method shared object file name is "lib*class-name*_SET_*property-name*" (*), you can omit the method entry information.

*: Specify 2 underscores between class name and GET or SET.

Note

If the class or method source program is compiled using the compile option ALPHAL, the class and method names are non case-sensitive. For details, refer to "A.2.1 ALPHAL (lowercase handling (in the program))".

At this time, use upper-case letters for the entry information class and method names.

Required Entry Information for Execution

When you execute an application, the entry information that is required for programs, classes, and methods compiled in that application using the DLOAD compiling option is shown in Table below.

Table 15.3 Required entry information for execution

Type	Class entry information	Method entry information
Program	Class described in the procedure division	Dynamic program structure method in the called method
Method		

Type	Class entry information	Method entry information
Class	Class described in the procedure division The class specified in the USAGE OBJECT REFERENCE clause of the returning item definition in the linkage section of the method definition(*)	

* : This class entry information is required for enabling the conformance check when executing the original method call in the source program. For details about the conformance check at the time of execution, refer to the CHECK(ICONF) explanation in "13.4 Conformance", "A.2.5 CHECK (whether the CHECK function should be used)".

Example

The entry information required for executing program P1 is explained. In all the following source programs, assume that the compile option DLOAD has been specified.

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

```
[CLASS]
C1=libC1.so
C2=libC2.so
C3=libC3.so

[C1.METHOD]
M1=libC1_M1.so

[C2.METHOD]
M2=libC2_M2.so
```

libC1.so

```
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.
```

libC2.so

```
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.
```

libC1_M1.so

```
METHOD-ID. M1 OF C1.
        :
END METHOD M1.
```

libC2_M2.so

```
METHOD-ID. M2 OF C2.
        :
END METHOD M2.
```

libC3.so

```
CLASS-ID. C3 INHERITS FJBASE.
        :
END CLASS C3.
```

_: This is the class or method required for entry information

Information

In the above entry information, the class shared object file name is "libclass-name.so", and the method shared object file name is "libclass-name_method-name.so". For this reason, you can omit them.

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.

Table below explains the resources you need to disclose to share classes with other developers:

Table 15.4 Resources to Disclose

	Resources name	Use
[1]	Document	It is necessary to tell functions of classes, interface (method name, parameter, and property name, etc.), and necessary resources (repository file name and shared object name, etc.) to the users of the class.
[2]	Shared object file (Object file)	Required when classes or programs are to be linked with a dynamic link structure or dynamic program structure. (When classes are created by a simple structure, the object files are needed.)
[3]	Repository file	Required if the classes or programs are to be compiled.

Information

In document to publish the classes, at least the following contents should be described:

- Shared object file names.
- Class names that include classes to inherit and the overview of functions.
- All method names and the overview of function.
- All property names and the overview of function.
- Interface of methods or properties (parameter and return value).
- Repository file names.
- Other notes.

15.9 Cautions for Executing Programs

This section explains the cautions for executing programs in the Object-Oriented programming.

15.9.1 Stack Overflow

In object-oriented programs, more stacks are used than used previously in conventional programming. Therefore, enough attention must be paid to the stack overflow. For more information, refer to "4.5.1 If a Stack Overflow Occurs During COBOL Program Execution".

15.9.2 Blocking Object Instances

This section explains how to block object instances in COBOL object-oriented programs.

15.9.2.1 Overview

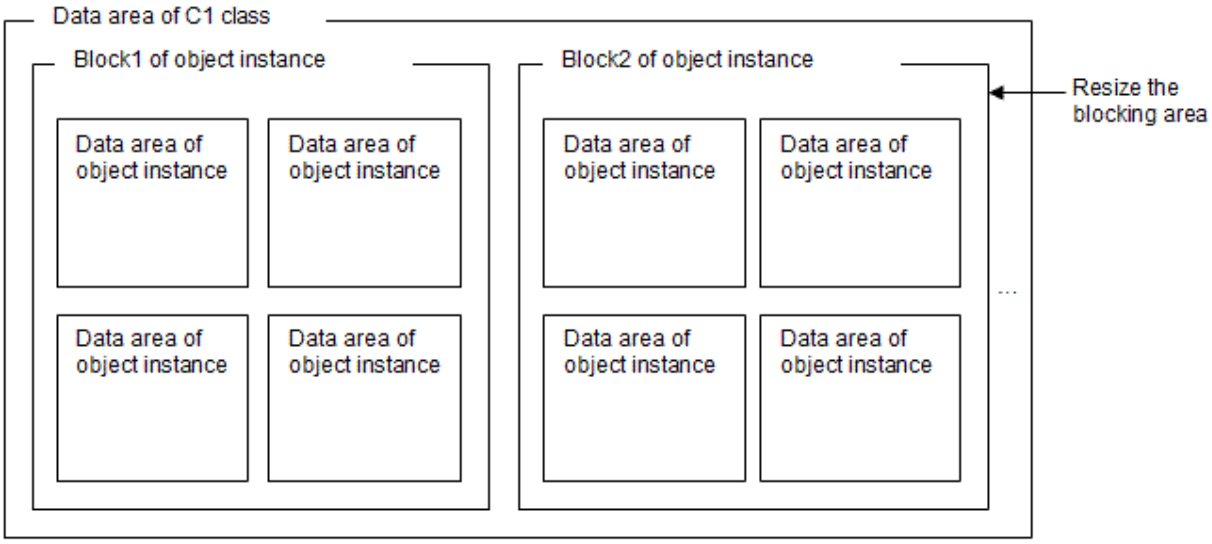
In object-oriented programs, object instances are generated by executing the NEW method. Then a work area needed for operating data items and object instances described in the data section of the object instance are acquired as the data area for the object instance.

The size of the data area of the object instance is different depending on the inheritance relationship of the class that includes the contents of the object definition described in COBOL source program and the object instance to be generated.

The data area of the object instance is usually made into blocks in the timing of the execution of the NEW method and acquired. Blocking means acquiring data areas for 2 or more object instances at a time.

Area length is not changed in the execution because the data area of the object instance that the class needs is a fixed for each class. However, size of a data area that an object instance has been made into blocks can be decided in the execution.

Blocking the object instance is to construct the most appropriate memory environment by controlling how to acquire data areas of the object instances that includes blocking. Specifically, memory to be used can be saved and the execution performance can be improved by setting the method of acquiring a correct object instances for the use of the classes in the applications.



15.9.2.2 Saving Memory

To save memory use, data areas of object instances acquired while executing applications should be minimized.

If there is no specification in the data area of the object instance, the COBOL runtime system considers the execution performance and executes the blocking processing in the value decided to be the best. However, areas that may not be used may have been acquired by blocking depending on the behavior of the applications. Therefore, the memory use can be reduced because only necessary areas are acquired by blocking the data area of the object instances.

To acquire the area without blocking the data area of the object instance prioritizing memories, specify as follows. For information about the specification of class information and the environment variable, refer to "15.9.2.4 Execution Environment Information on Tuning Memory".

Specification for class (Class information)

Specify 1 for the number of storage of object instances of the class to save memory use.

Class information file

```
[INSTANCEBLOCK]
C1=1
```

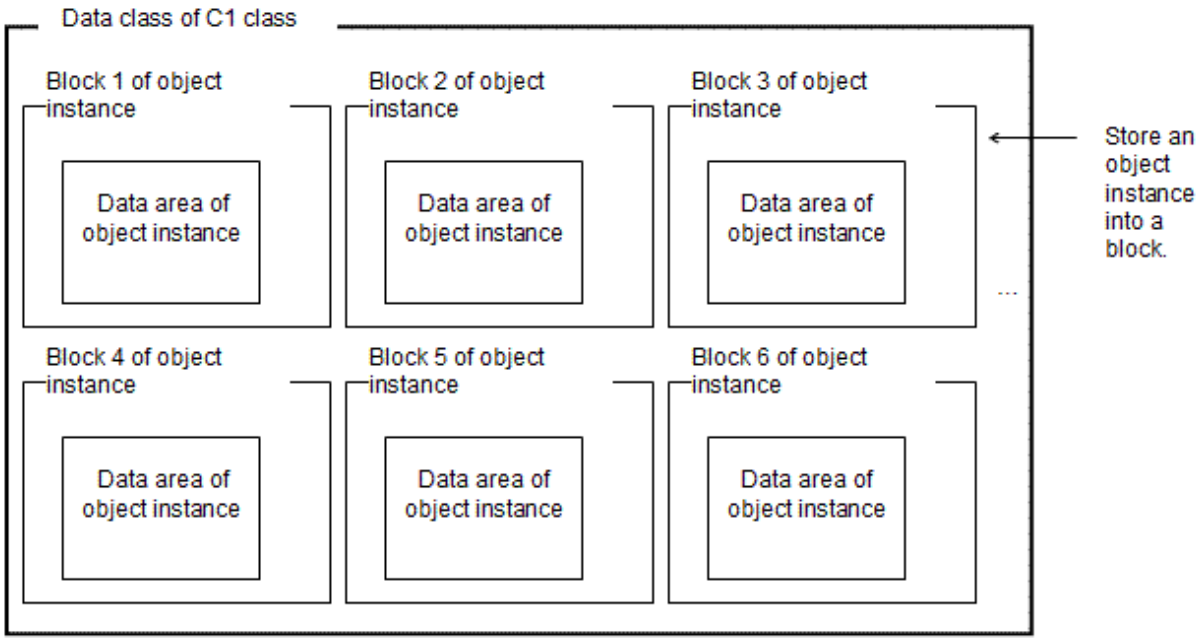
Specification for Run Unit of Application (Environment variable)

Specify to acquire object instances without blocking.

```
CBR_INSTANCEBLOCK=UNUSE
```

The minimum area needed in the object instance is acquired by specifying either of the above-mentioned specifications. Mainly when the data area described in the data part of the object definition is large, it is effective in the case where the hierarchy inherited by the class is deep.

However, the execution performance decreases depending on the applications because the specification enables them to acquire an area in each generation of object instances.



15.9.2.3 Improving Execution Performance

To improve execution performance, data areas of object instances should be efficiently blocked according to the purpose of the applications. When blocked, the needed areas are allocated and used from the acquired areas at the time of object generation because the data area of the object instance that can be used in the future is acquired at that time. As a result, the overhead of the area acquisition processing is reduced and the execution performance improves depending on the specified values.

Follow the specification below in which unit the area of the object instances is acquired collectively when the applications are executed. For more information about the specification of class information, refer to "15.9.2.4 Execution Environment Information on Tuning Memory".

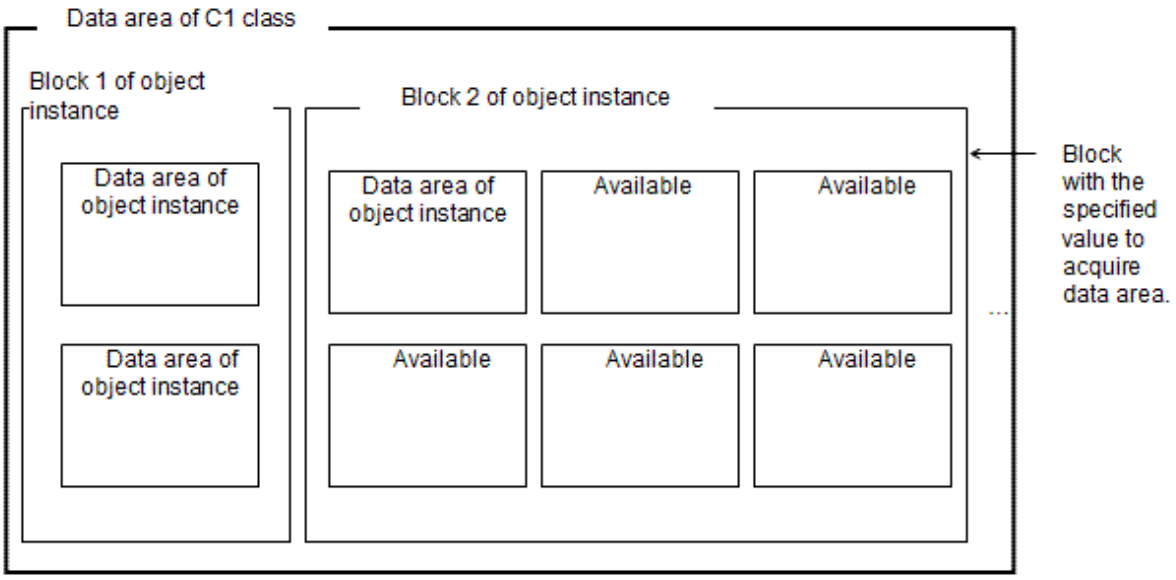
Specification of Each Class (Class information)

Specify an arbitrary number for the storage number (initial number and increment number) of object instances of the class to be blocked.

Class information file

```
[INSTANCEBLOCK]
C1=2,6
```

In the above-mentioned example, the areas where 2 data areas of the object instance can be stored when the object instance for "C1" class is first generated while executing the application are acquired. After that, if data areas allocated in the areas acquired in the first acquisition do not exist when the 3rd instance is generated, the data areas of the object instance acquires 6 data areas that can be stored in addition.



If there is an area reusable in the area of the blocked object instance, the area is allocated in the new object instance when the generation and deletion of the object instance in the same class are repeated.

15.9.2.4 Execution Environment Information on Tuning Memory

This section explains execution environment information on tuning memory.

15.9.2.4.1 Environment Variables

The list of the environment variables is shown below.

Relating to Files

CBR_CLASSINFFILE (Specifying class information file)

```
CBR_CLASSINFFILE=Class-information-file-name
```

Specify a class information file name by which the class information to be used in an object-oriented program is defined. For more information about class information, refer to "15.9.2.4.2 Class information".

An absolute path and relative path can be specified for class information files. When a relative path is specified, it becomes a relative path from the directory where the executable file under execution exists.

Relating to Object Instances

CBR_INSTANCEBLOCK (Specification of method of acquiring object instance)

```
CBR_INSTANCEBLOCK=[{USE,UNUSE}]
```

Specify USE to block areas of object instances of each class and acquire them or UNUSE not block the areas. When UNUSE is specified, the minimum area on each timing of the generation of the object needed in the object instance is acquired. This specification is enabled for the all classes of COBOL used by object-oriented programs. However, the class that the storage number of object instances has been specified in the class information file blocks and acquires the area of the object instance according to the value specified for class information.

15.9.2.4.2 Class information

Specify information on the classes to be used in object-oriented programs in each section. Information to be specified is shown below.

Class Information

A text file that stores the class information is called a class information file. For information about how to specify the class information file when the program is executed, refer to "15.9.2.4.1 Environment Variables" in this chapter.

INSTANCEBLOCK Section (Specification of the number of storage of object instances)

```
[INSTANCEBLOCK]
Class-name=Initial-number [,incremental-number]
```

Specify the number of object instances to be stored in the area of the object instance acquired when the program is executed in initial value and incremental value. The values that can be specified for both initial values and incremental values are an integer of 1 or more. When specifying incremental value is omitted, it is considered that 1 is specified.

When the area of the object instance is first acquired, the area where the data area of object instances of the number specified in an initial value can store is acquired. When the object instance that exceeds the number specified in the initial value is generated while executing the program, the area where the data area of object instances of the number specified in the incremental value can store is acquired.

When there is no specification for the classes, follow the specification of the environment variable CBR_INSTANCEBLOCK.

For more information, refer to "15.9.2.4.1 Environment Variables".

Chapter 16 Multithread Programs

Creating multithread programs enables COBOL programs to be executed in multithread environments and is specific to server-oriented operation environment products.

This chapter explains how COBOL programs are executed in multithread environments.

16.1 Overview

If a COBOL application is used as a server application by using functions such as Web and distributed objects, many clients request its execution, thereby causing the server load to be sharply increased. Applying the multithread function can reduce used quantities of resources used, such as server memory and hard disks. Moreover, execution performance can be improved.

16.1.1 Features

Use of existing COBOL resources in multithread environments

Existing COBOL resources can be used in multithread environments only by recompiling the programs.

It is not a problem to migrate or shift COBOL resources to server products that call the programs under a multithread environment. Server products that call the programs under multithread environment include the following:

- Application server products that use distributed object technology (i.e. CORBA).

  Class definitions sorted in more than one environment on the server via application servers can be executed by calling the class definition from the COBOL application on the client. This allows you to sort class definition by thread under multithread environments.

- Web Server products.

  The browser on a client sends a request to the Web server via a Web link. Upon receiving the request, the server executes a registered COBOL application. More than one request in separate threads can be allocated under multithread environments.

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

you can use the COBOL-provided TRACE, CHECK, and COUNT functions and system standard debugger to debug programs operating in a multithread environment. You can also use the subroutine for obtaining process IDs and thread IDs to help you in debugging.

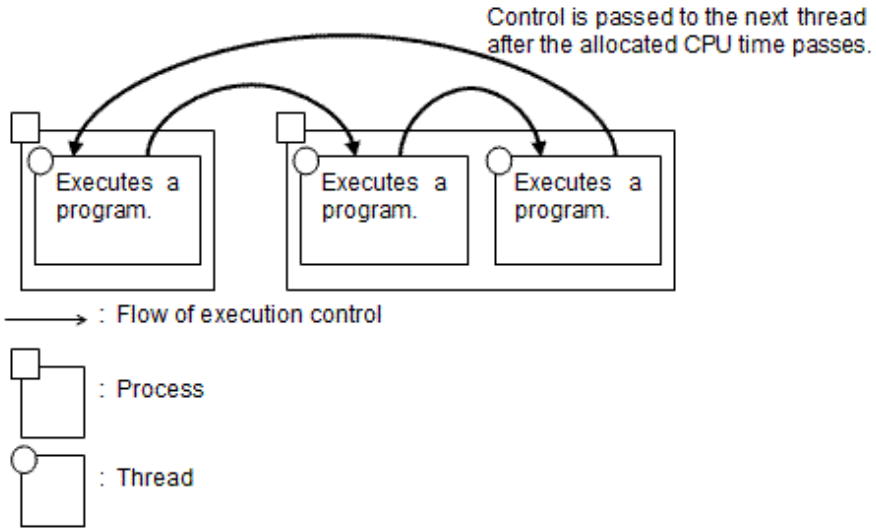
COBOL provides no function for activating threads.

16.2 Multithread Advantages

16.2.1 What is a Thread?

A thread is a minimum run unit the schedule of which is controlled by an operating system. The operating system schedules a thread to be executed and its execution time, and allocates CPU time to it. After the allocated CPU time has passed, 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.



16.2.2 Multithread and Process Model

System library to enable the multithread function is linked to the applications that run under the multithread environments. The library is not linked to the applications under legacy process environments. You need to prepare programs of models that are executable under each environment because the system library to enable the multithread function changes the system running environments. It is the multithread program and process model program.

When called from linked products such as Web Server under multithread environments, the COBOL programs that are called should be multithread models. The process model program should be executed when called from a linked product under the process environment.

Process Model Program

The process model program can be executed only in a thread in a process. Therefore, the same program cannot be executed at the same time in plural threads in a process and different programs cannot be executed concurrently either.

To create a COBOL process model program, an object file compiled for the process model and COBOL runtime systems that correspond to the process model should be linked.

Multithread Program

Multithread programs can execute a COBOL program in plural threads in a process or with multithreads.

To create a COBOL multithread program, an object file compiled for the multithread model and COBOL runtime systems that correspond to the multithread model should be linked.

Note

Multithread programs and process model programs cannot be executed at the same time.

16.2.3 Multithread Efficiencies

The efficiencies given below can be implemented by compiling server applications, which use server-activated functions such as Web and distributed objects, as multithread programs:

High-speed startup.

When a process model program is executed more than once, initial processing in the process for the executed times are executed. On the other hand, when the same program of the multithread model is executed in more than one thread, initial processing after the second time is unnecessary and only initial processing of the thread for the executed times is executed. Because the time for the initial processing of the thread is shorter than that of the process, the boot time is consequentially shortened.

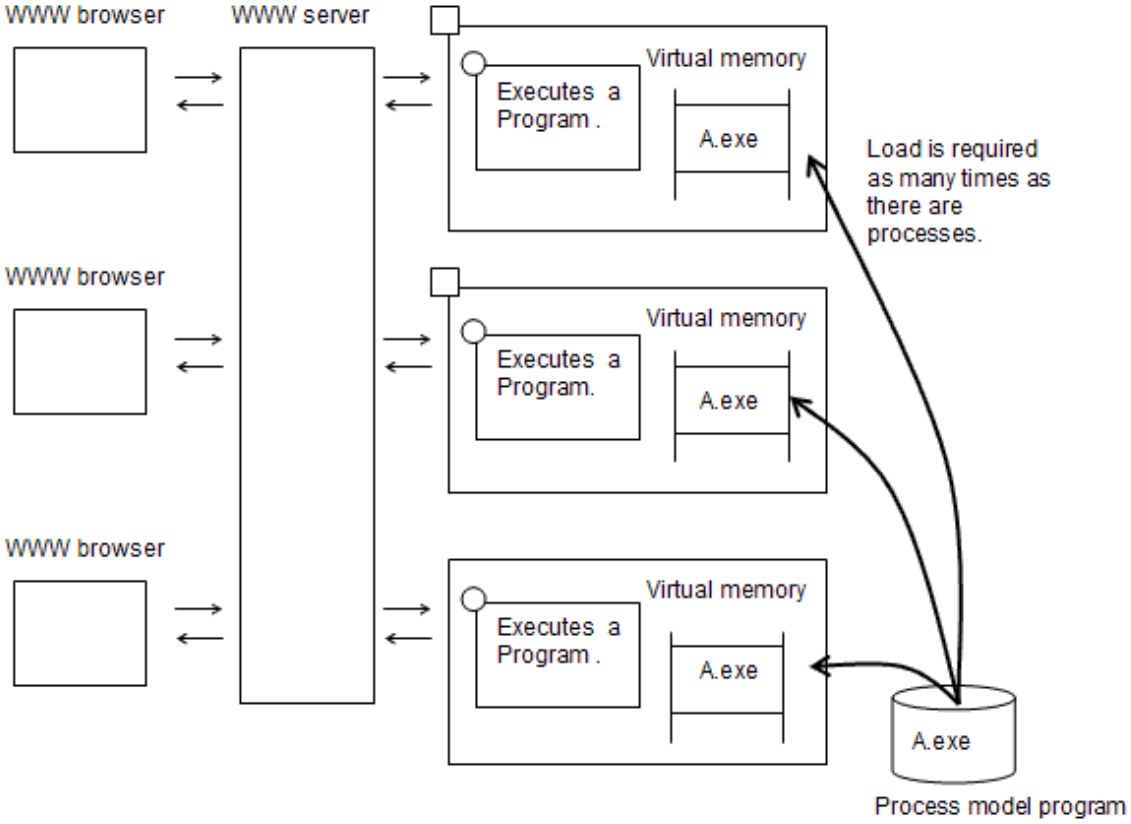
Multiple operation.

In multithread programs, memory can be saved because the process space is shared by all threads in the process. Therefore, even if the multiple programs runs concurrently, the load to the system is reduced compared with the process model. In other words, on the same memory environment, the multithread model can operate more programs at the same time.

The following example explains the case where a Web server is used and the effect of the multithread program:

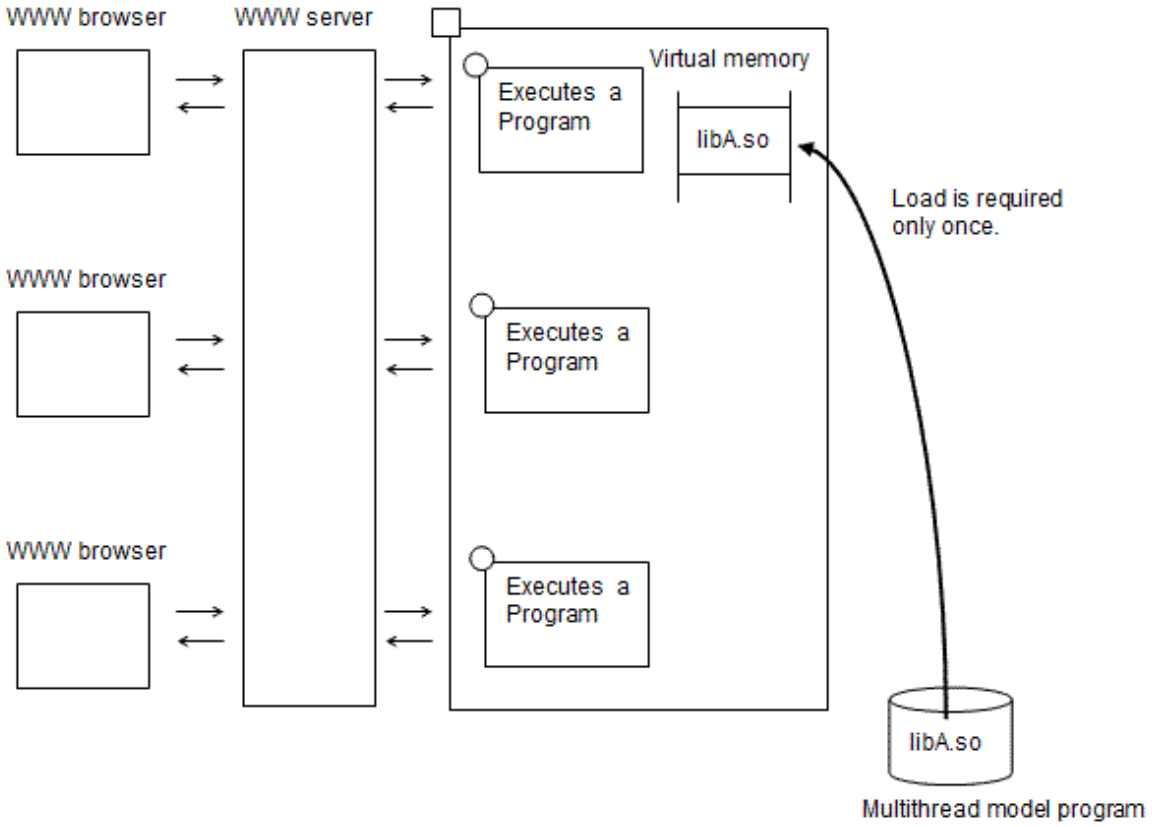
Process Model Program

The process model-base server applications include such as the Web application like Common Gateway Interface (CGI). When a Web server receives a start request of CGI application from a client, it starts the process model program that is the execution form file as a new process. To start a process, processing such as acquisition of process space, loading load modules and initializing data are necessary. Also, the necessary amount of memory for each process will be acquired.



Multithread Programs

When a server based on the multithread model is used, the server does not start a new process but executes a multithread program by one thread in the process. A multithread program is a common object file and it is loaded into process space when called for the first time and usually resides in the system afterwards. Therefore, start-up processing is shortened and the amount of necessary memory is reduced.



16.3 Operation of COBOL Programs

This section explains the differences in behavior between the COBOL process model program and that of the multithread model.

16.3.1 Runtime Environment and Run Unit

As described in "8.1.1 COBOL Inter-Language Environment", COBOL supports runtime environments and run units. In multithread program, a runtime environment is supported for each process, and a run unit is supported for each thread. The following section explains the multithread model runtime environment and run unit.

Runtime Environment

The runtime environment of a COBOL multithread program is set up when it is called by a process for the first time. It is closed when the process terminates or JMPCINT4 is called. Similarly to a process model 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.

These resources are the follows:

- Factory object.
- Object instance.
- System standard input and output.
- Files used by the ACCEPT/DISPLAY statement.
- External data/file shared by threads.

JMPCINT4

This subroutine is provided to close a 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 in Linkage with another Language" for the calling format of JMPCINT4.

Run Unit

The timing of starting and terminating a run unit of a multithread program is the same as that of a process model program. Because a COBOL program is executed by two or more threads, 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), and so on.

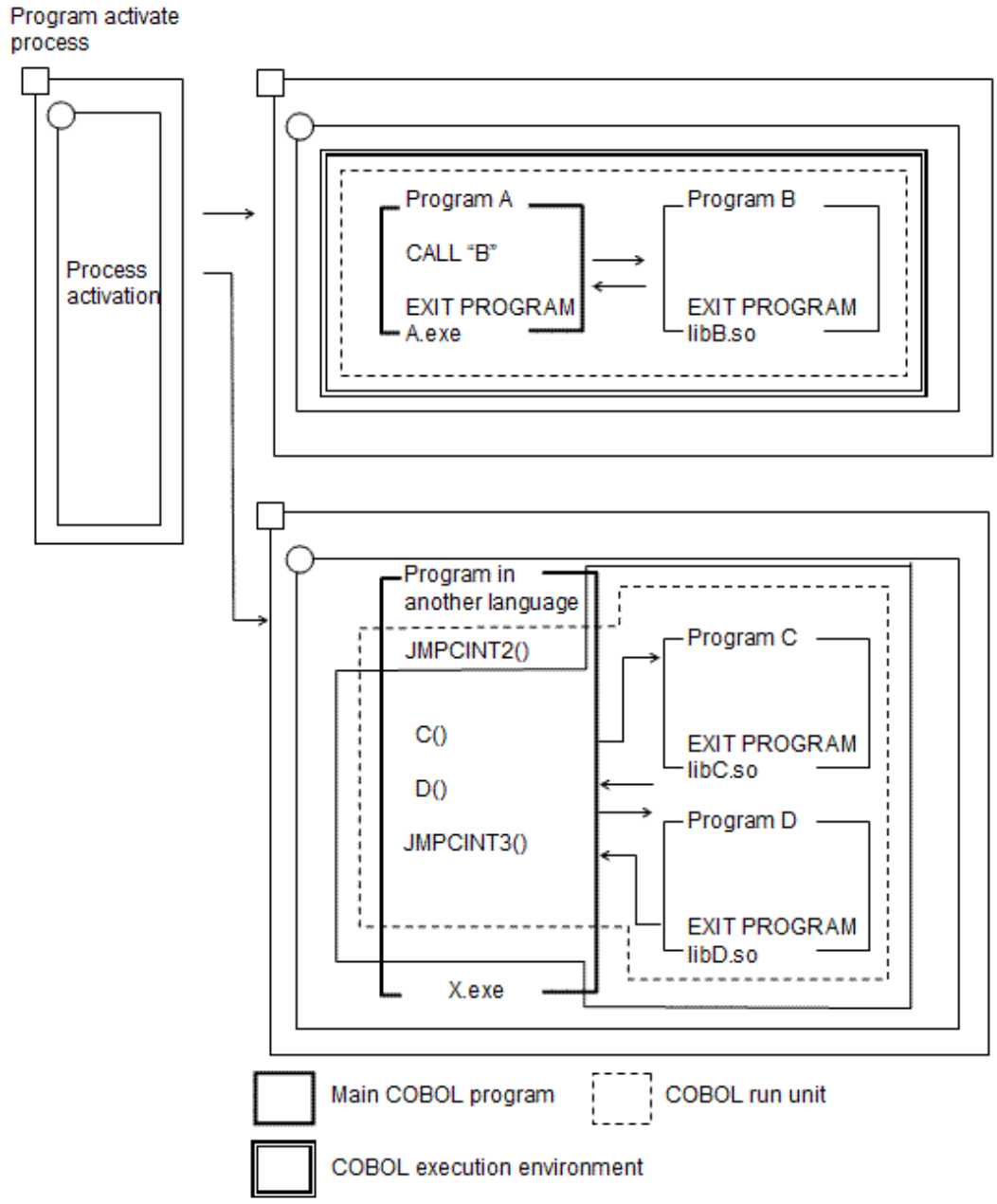
Be sure to call JMPCINT3 when calling JMPCINT2. Otherwise, the COBOL run unit will not terminate and the resources used by the thread will not be released, thereby causing a malfunction.

For more details, refer to "G.2 Subroutines Used in Linkage with another Language".

The figures given below show the relationships between the runtime environment and run unit of process model programs and multithread programs. For a process model 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.

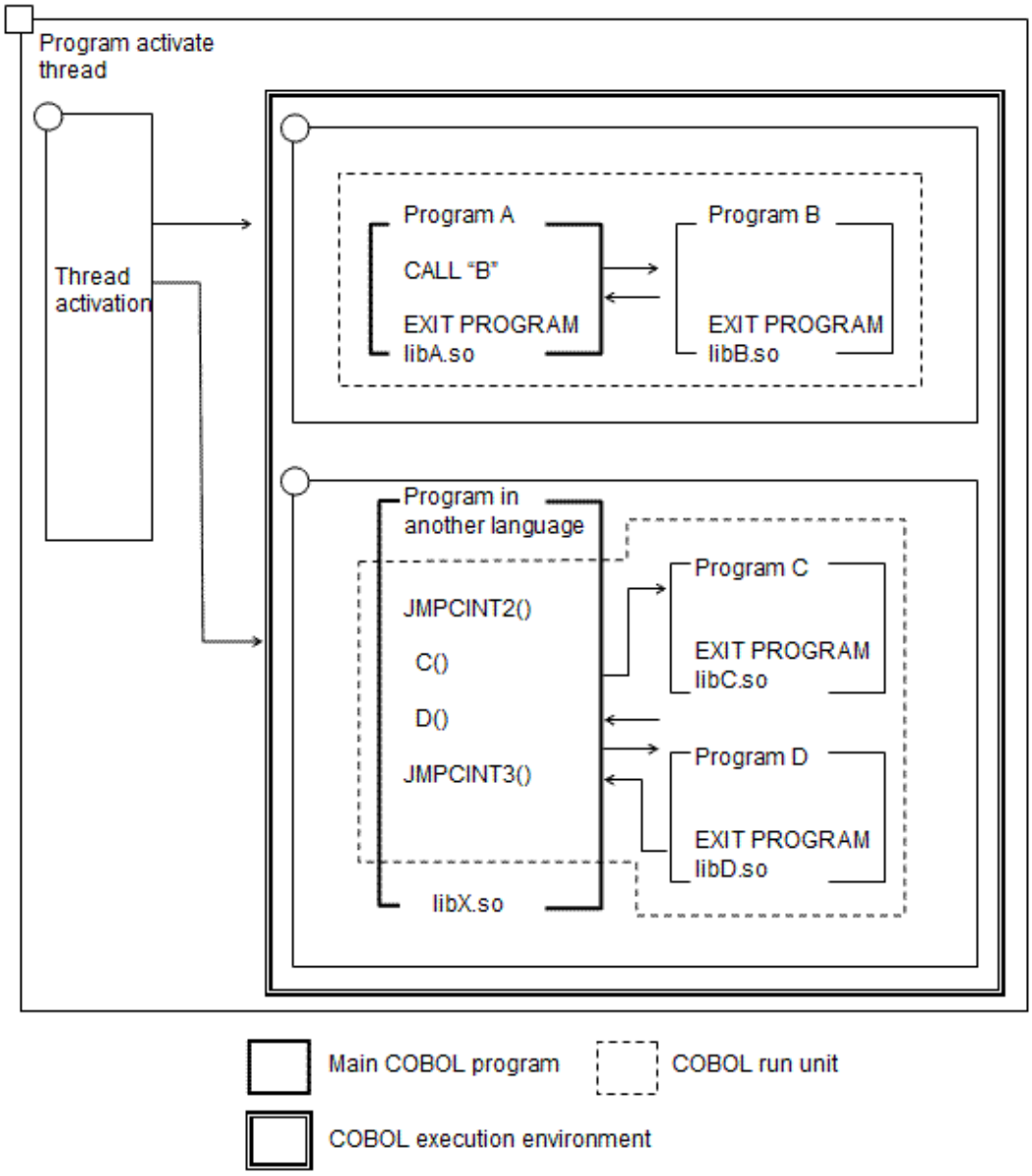
Process Model Program

For a process model 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.



16.3.2 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.
- Data Acquired and Managed By Thread
  - Data declared in program definitions. (*2)
- Data Acquired and Managed By Calling Unit
- Data declared in method definitions

*1: Data or files for which the EXTERNAL clause is specified in COBOL source programs compiled with specification of compiler option SHREXT when creating multithread program.

*2: External data and files shared among threads are excluded.

The following section describes these data items.

16.3.2.1 Data Declared in Program Definitions

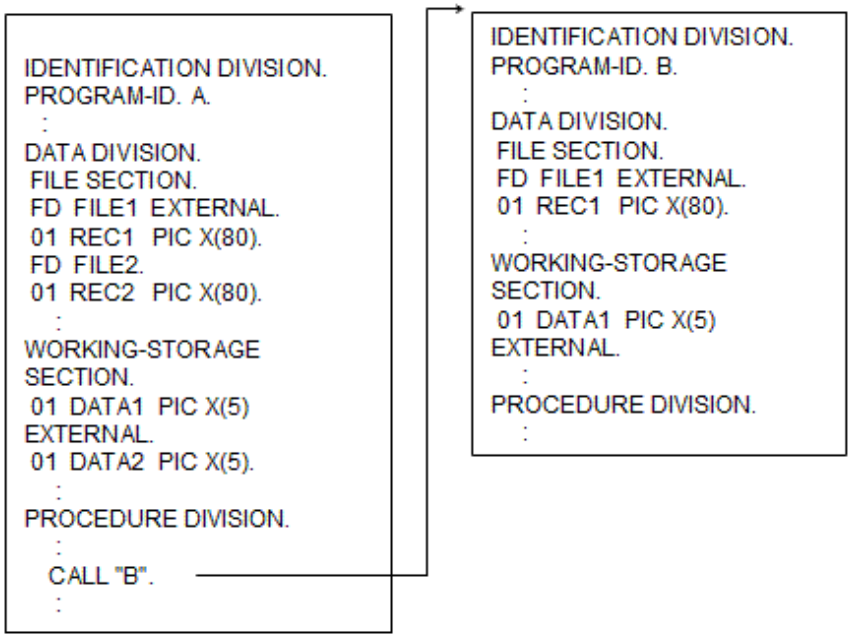
Data declared in program definitions is allocated for each thread.

This area is allocated when a run unit starts, and is closed when the run unit terminates. Files and cursors not closed at termination of the run unit are forcibly closed.

Note

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.

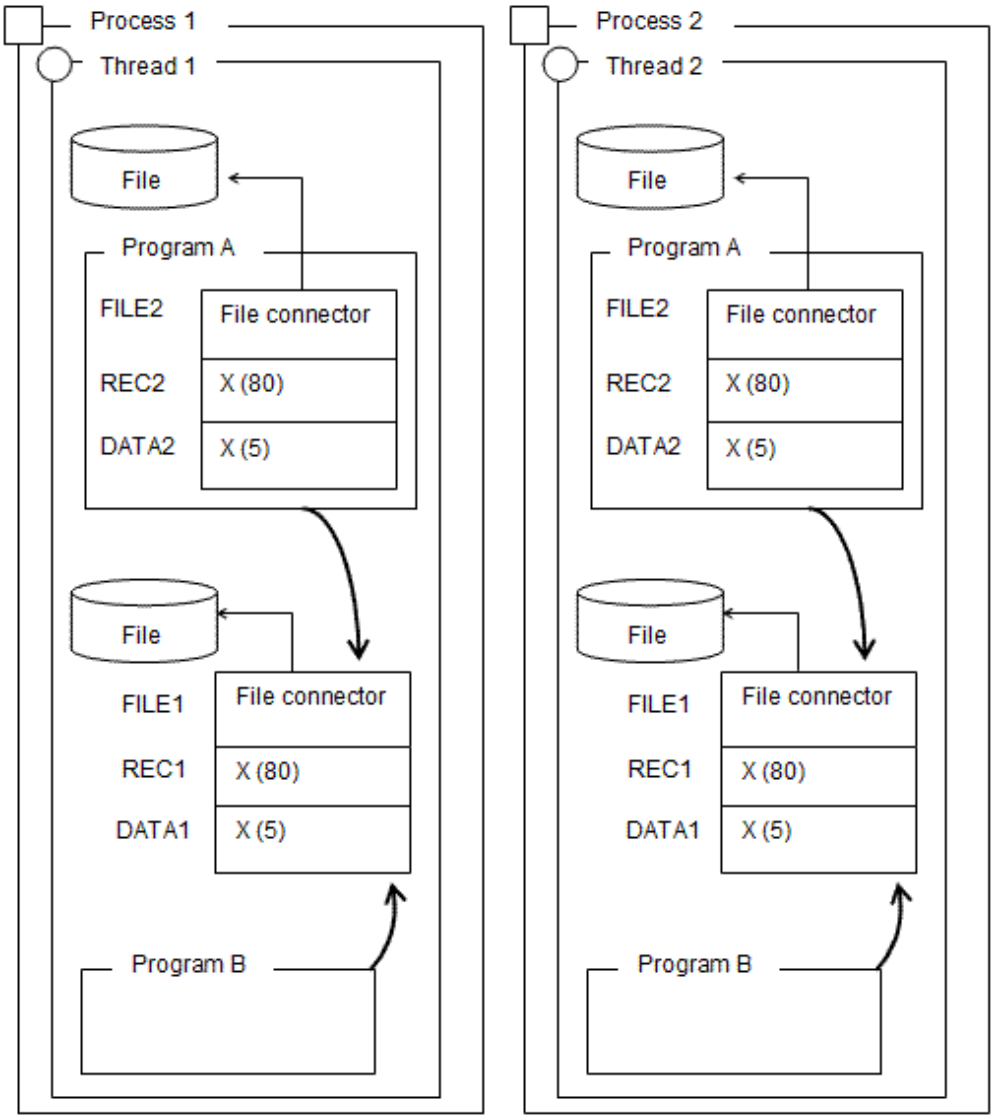
Data and Files Declared in Program Definitions



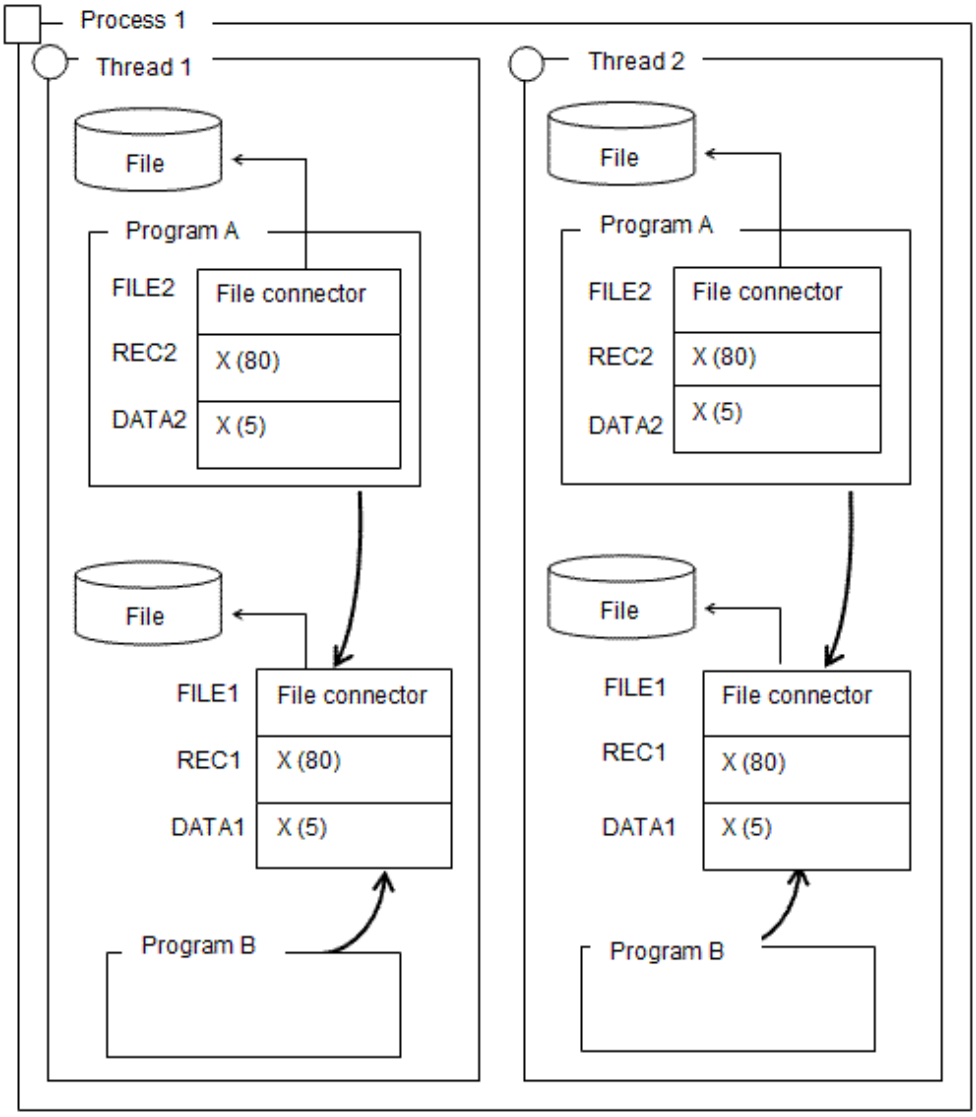
The figure given below shows activation of two of the above program. If the program is activated as a process model program, two processes are activated. If the program is activated as a multithread program, two threads are activated.

As shown in the figure below, for the process model program, data is allocated for each process, and for the multithread program, the data is allocated for each thread:

Process Model Program



Multithread Program



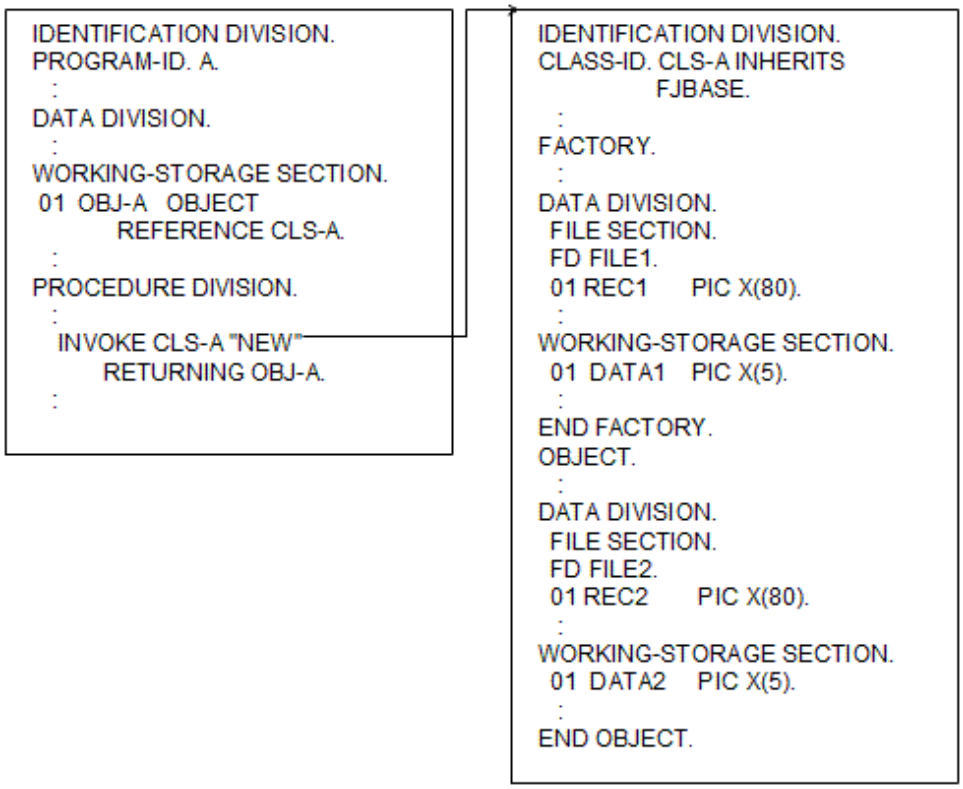
16.3.2.2 Factory Object and Object Instance

Factory objects and object instances are controlled by processes. Only one factory object of each class exists in one process; therefore, the object is always shared by threads in a multithread program. For more information, refer to "16.4.3.2 Factory Object".

Object instances can be shared by threads through factory objects. For more information, refer to "16.4.3.3 Object Instance".

Factory objects and object instances not released are released at termination of a runtime environment.

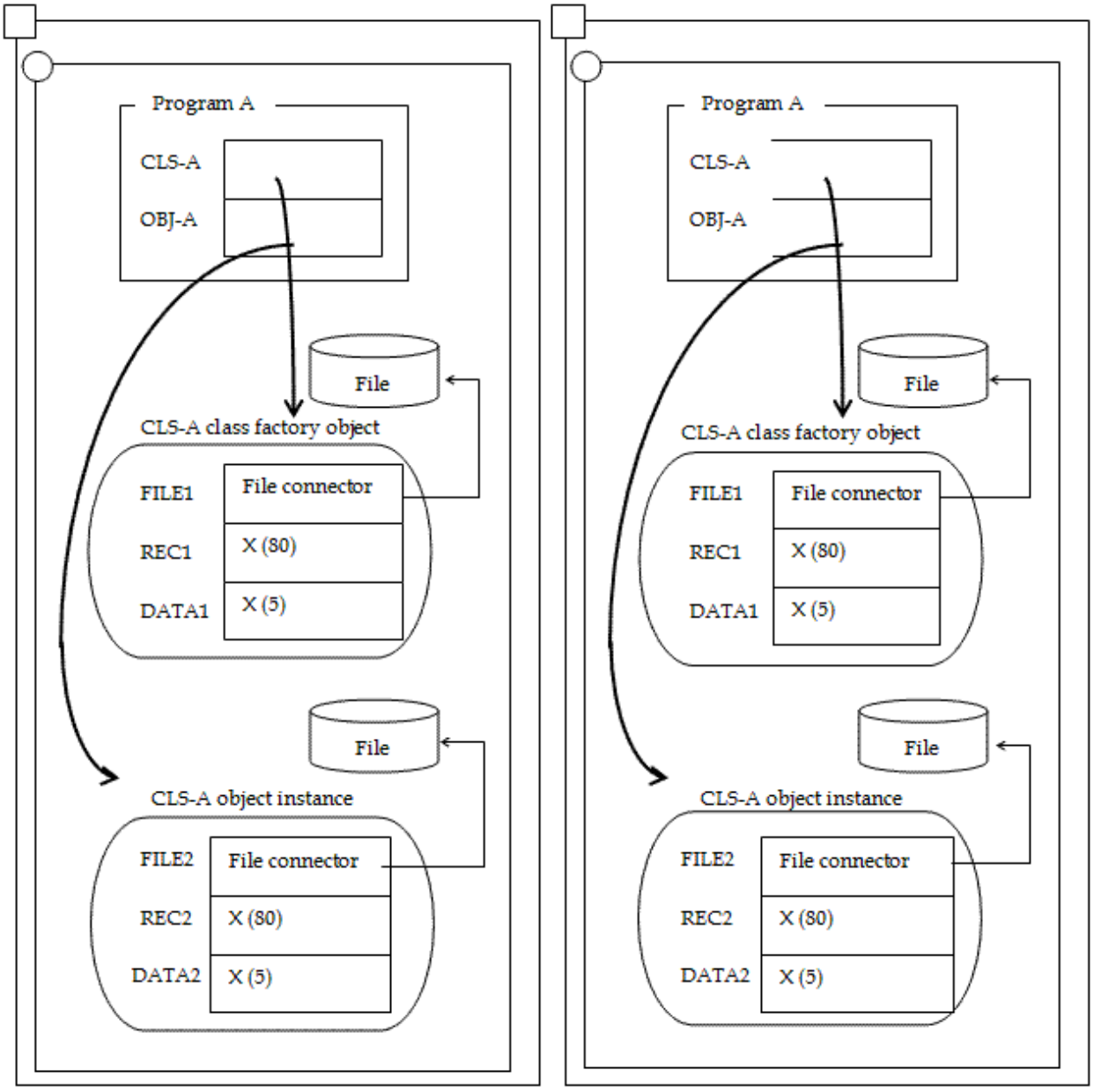
Data and Files Declared in Factory Definitions and Object Definitions



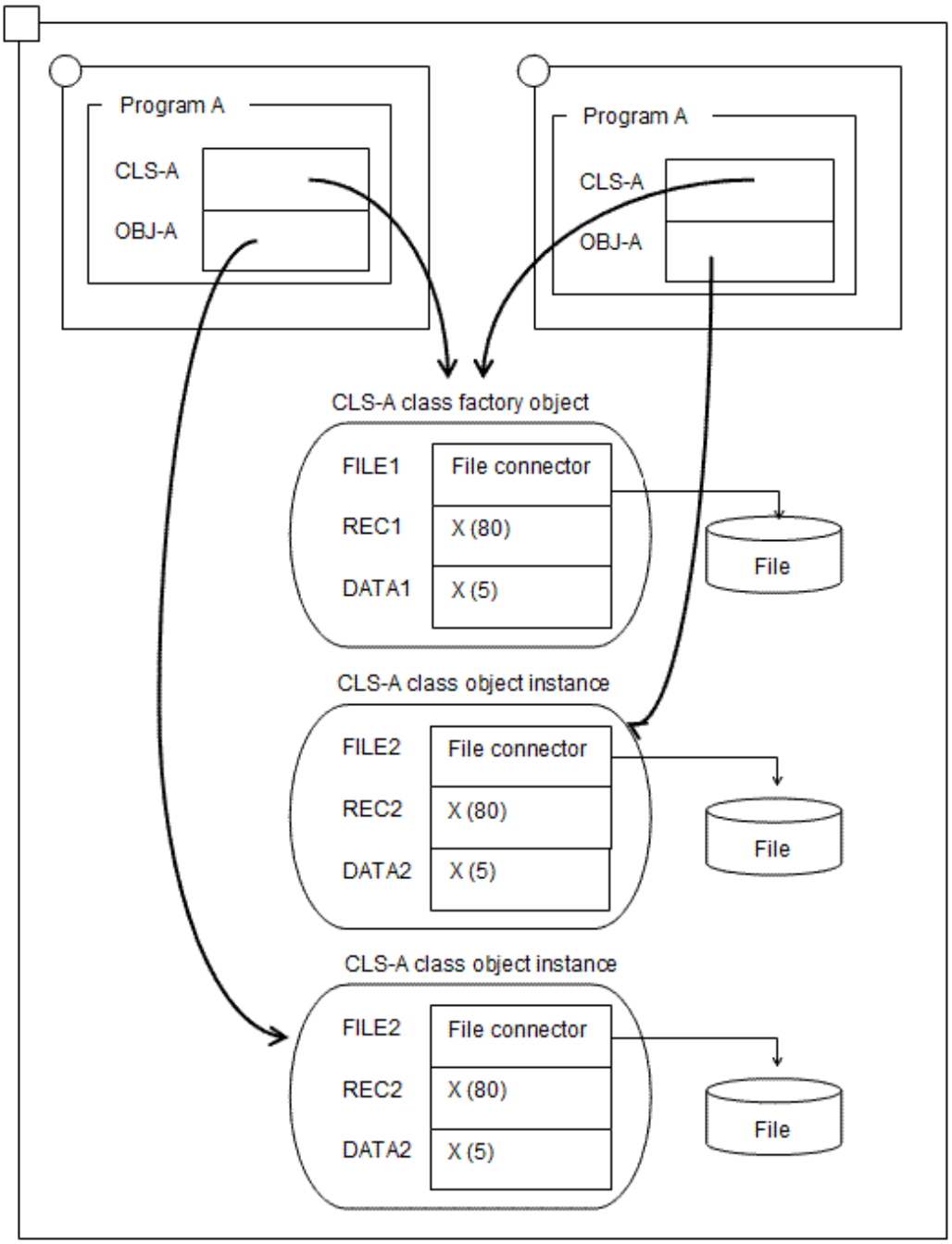
In the figure given below, two of the above programs are activated. If the program is activated as a process model program, two processes are activated. If the program is activated as a multithread program, two threads are activated.

As shown in the figure, the process controls the objects, in both the process model program and the multithread program. Therefore, in the multithread program, the factory object is always shared by threads.

Process Model Program



Multithread Program



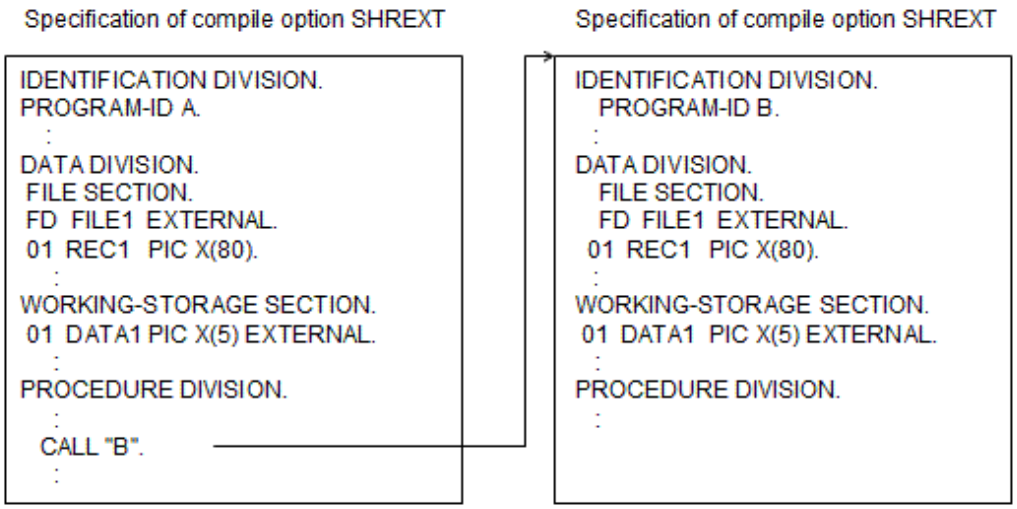
16.3.2.3 Data Declared in Method Definitions

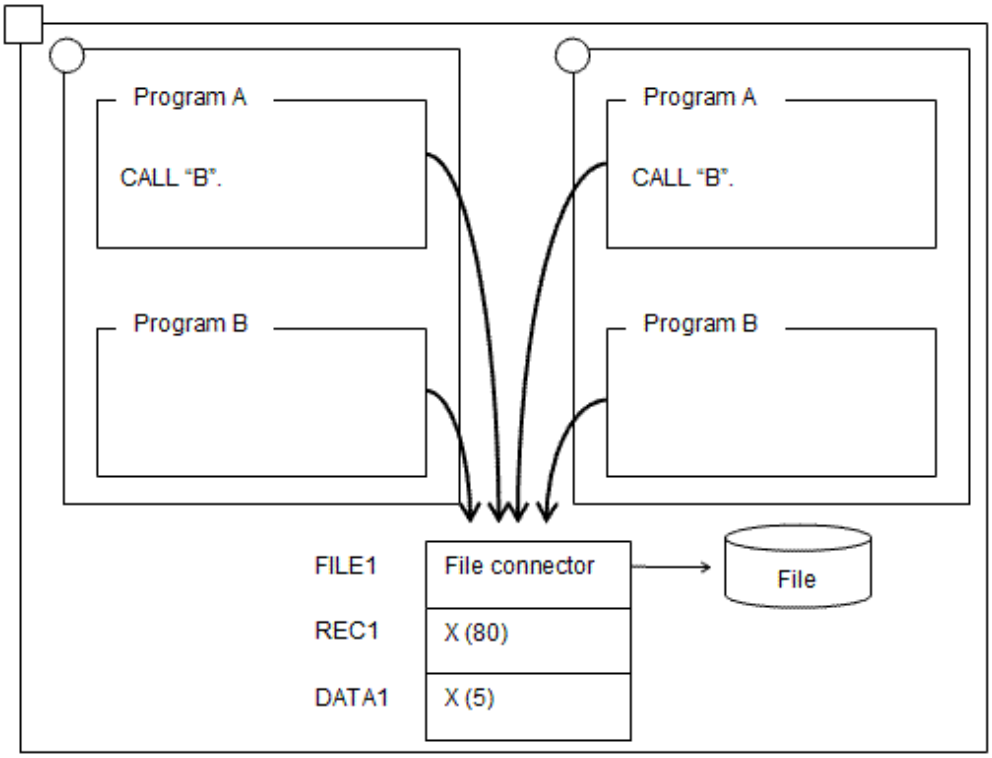
Data declared in method definitions is allocated using the same method as that for a process model program. That is, the data is allocated when a method is called and released when control is returned to the caller of the method. Files not closed are forcibly closed when control is returned.

16.3.2.4 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 when compiling the multithread program.

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.





16.4 Resource Sharing Between 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.

16.4.1 Resource Sharing

Resources cannot be shared among processes in the process model programs. On the other hand, resources can be shared among threads in multithread programs. A same file concurrently opened by programs that have been started separately by sharing the resources among threads can be accessed and linkage process between programs can be executed easily.

16.4.2 Race Condition

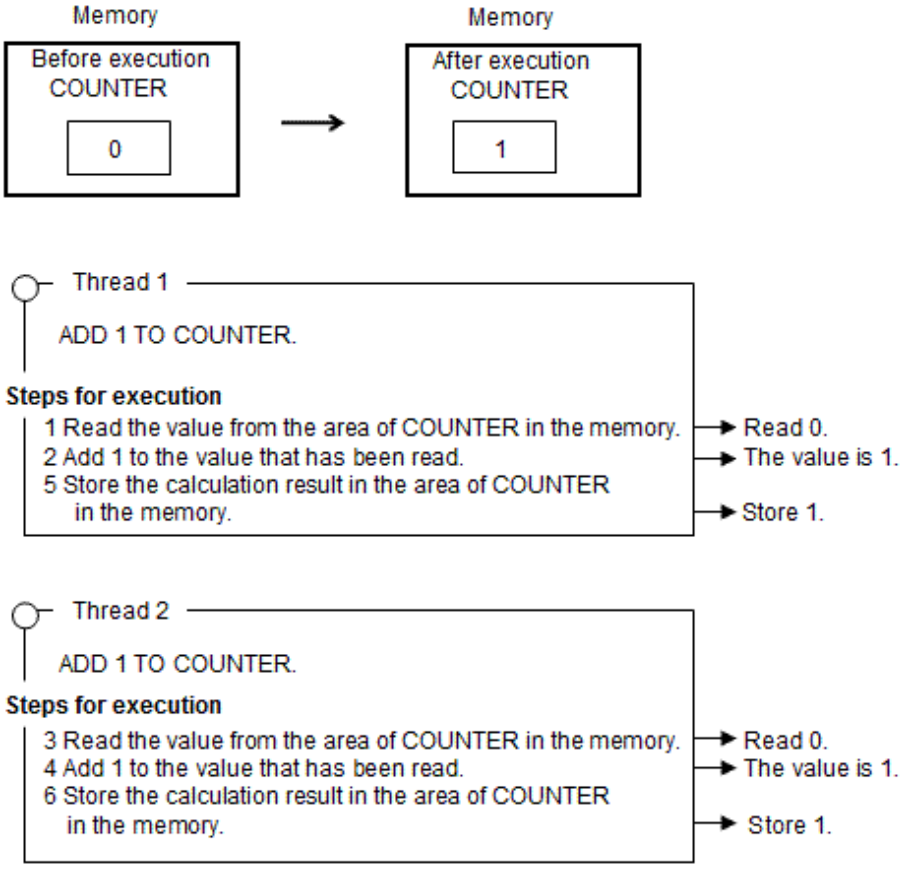
This section explains the race conditions that normally occur when resources are shared among threads.

The system cannot predict the order of executing threads because it compulsorily switches the threads to be executed. Therefore, when resources are shared among threads, the order of executing the threads might influence the result of the program. This is referred to as "race condition". The following example explains the race condition.

Assume that there are two threads that execute the "ADD 1 TO COUNTER" sentence for data named COUNTER shared between the threads. "ADD 1 TO COUNTER" is one sentence. It is compiled into some machine languages by a compiler and the system executes it in the following order:

1. Reads the value from the area of COUNTER in the memory.
2. Adds 1 to the value that has been read.
3. Stores the calculation result in the area of COUNTER in the memory.

As a result, when the order of executing thread 1 and thread 2 changes as follows, the value of COUNTER becomes 1 when the expected value is 2:



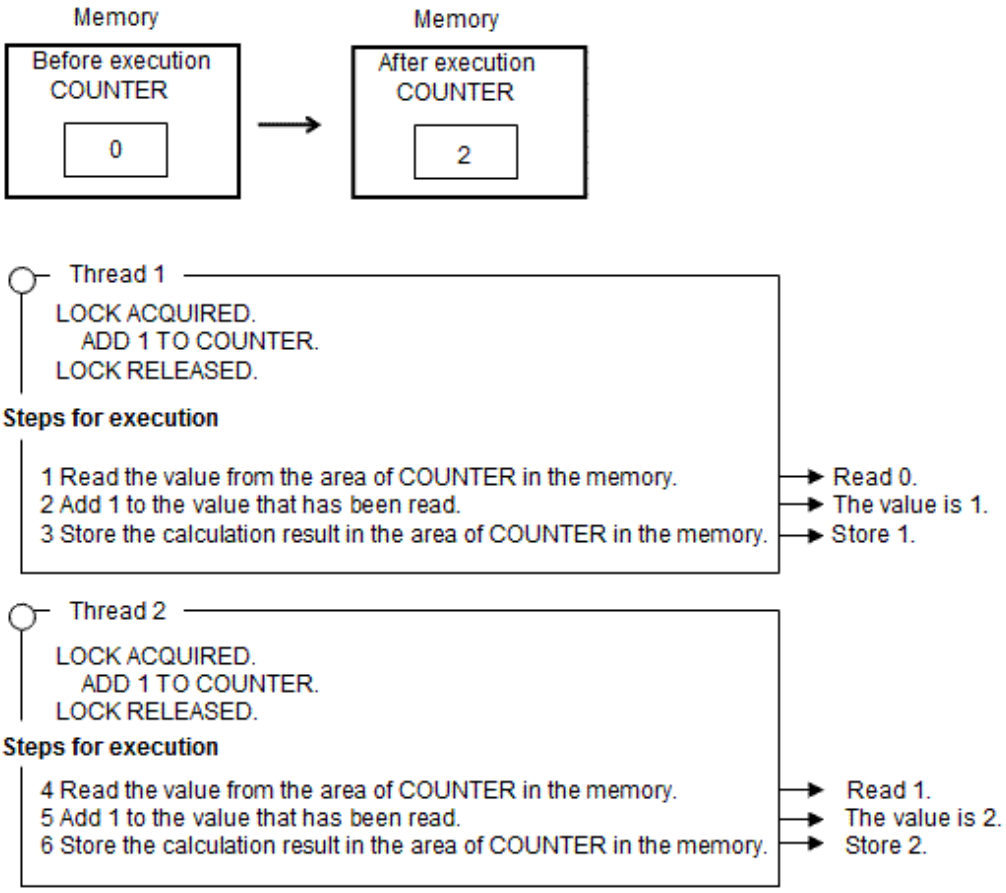
The following conditions generate a race condition. Of course, it is no problem for all the threads only to refer to the same data even if the data is accessed at the same time.

- If a thread that updates data and a thread that refers to the same data access the data at the same time.
- If a thread that updates data and a thread that updates the same data access the data at the same time.

Synchronization Control

Synchronization control is a mechanism for ensuring a continuously run series of procedure to shared access resources does not incur a race condition. Acquiring the execution right referred to as "Lock" guarantees continuous operation of the procedure with no race conditions occurring. Only one thread can acquire lock and only the thread that acquired the lock can be executed. Other threads that tried to acquire the lock will wait for the thread that acquired the lock to release the lock.

Because thread 2 is executed after thread 1 is executed by using the lock for the above-mentioned example, the value of COUNTER becomes 2. (In this example, it is assumed that thread 1 previously acquired the lock).



16.4.3 Resource Sharing in COBOL

In COBOL, the following resources can be shared among threads:

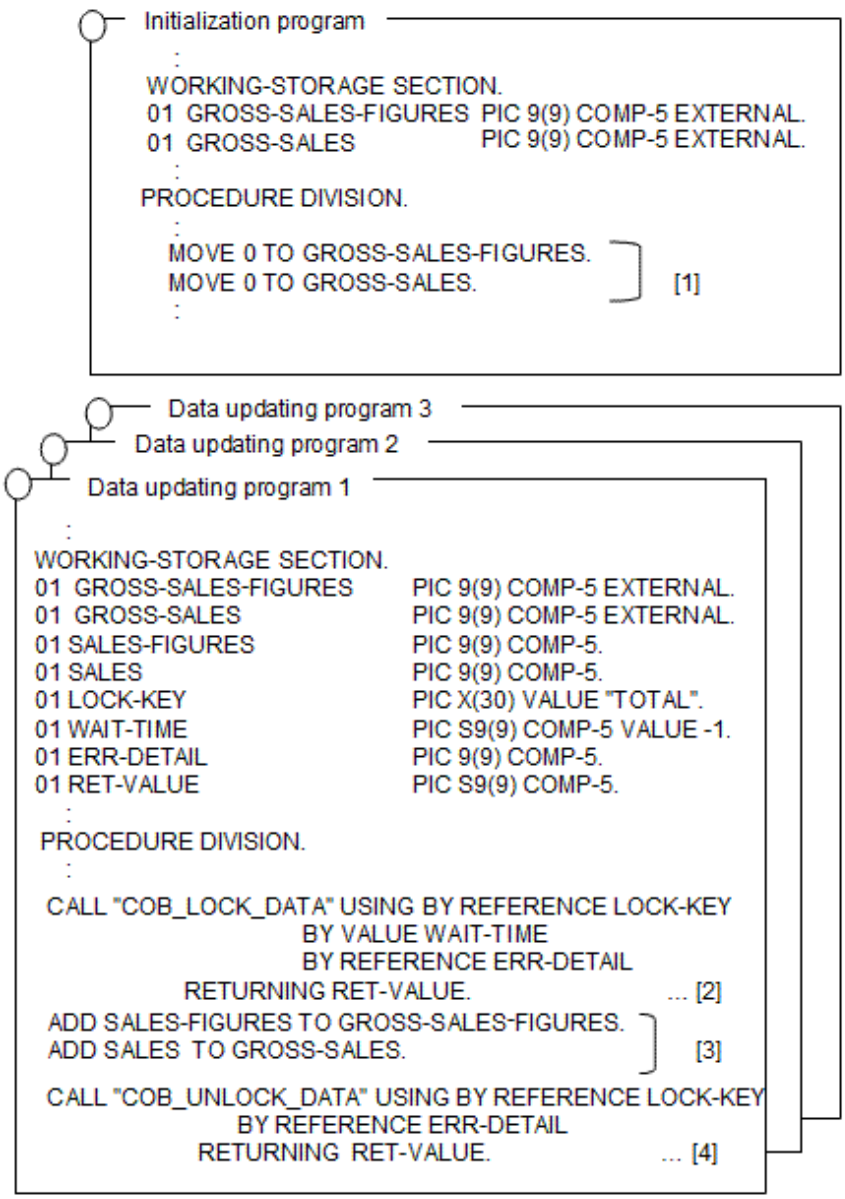
- External data and external file shared among threads
- Factory object
- Object instance

The COBOL runtime system performs synchronization control automatically for external files shared among threads for each input-output 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 "16.9 Thread Synchronization Control Subroutine".

16.4.3.1 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 when compiling the multithread program.

In the example given below, external data is shared among threads. Refer to "16.6.1.1 External File Shared between Threads" for information on sharing external files among threads.



Explanation of diagram

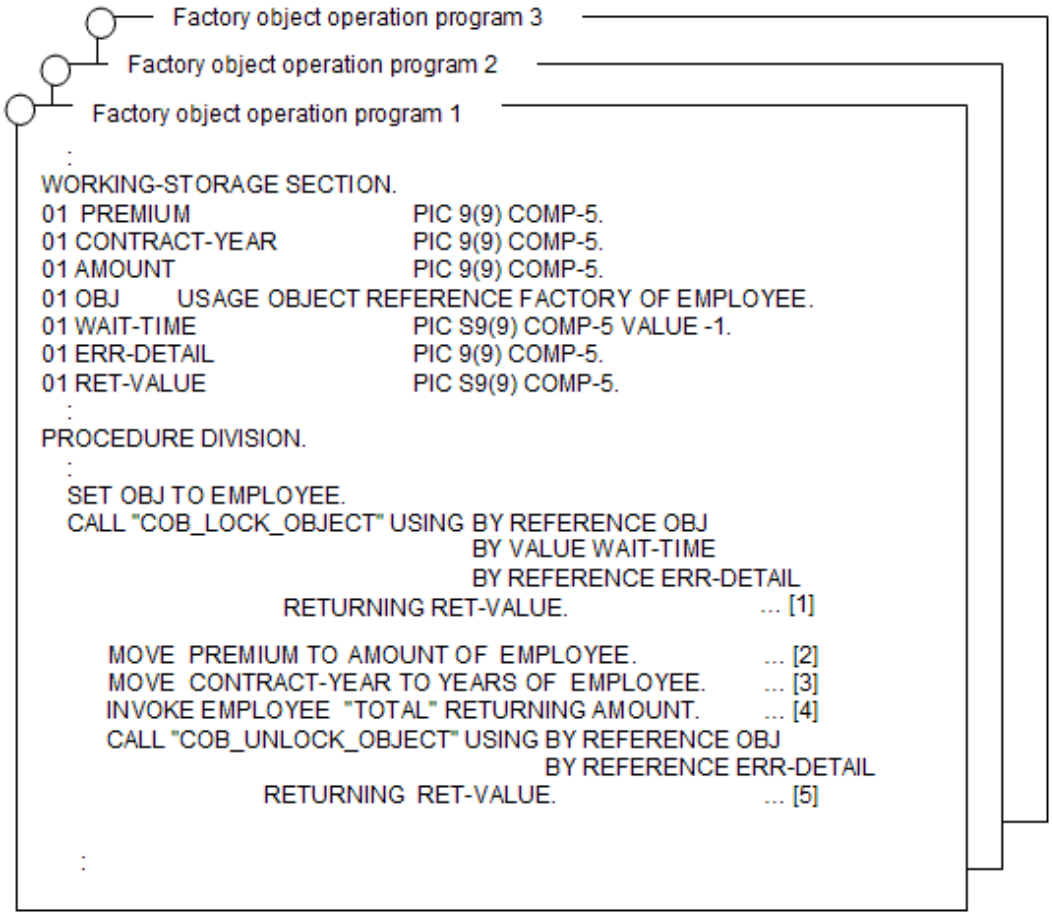
- [1] Shared data is initialized by the initialization program. The initialization program is activated first in an execution environment.
- [2] The lock for data name TOTAL is obtained by using the data lock subroutine to prevent the shared data from being accessed by two or more threads concurrently. Refer to "16.9.1 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 [2] .
- [4] The lock obtained for data name TOTAL is released. Another thread can obtain the lock by using the same processing as [2].

16.4.3.2 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 only by calling a factory method once, the program need not control thread synchronization.

However, if as shown in the example given below, processing is completed by calling methods more than once, 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.



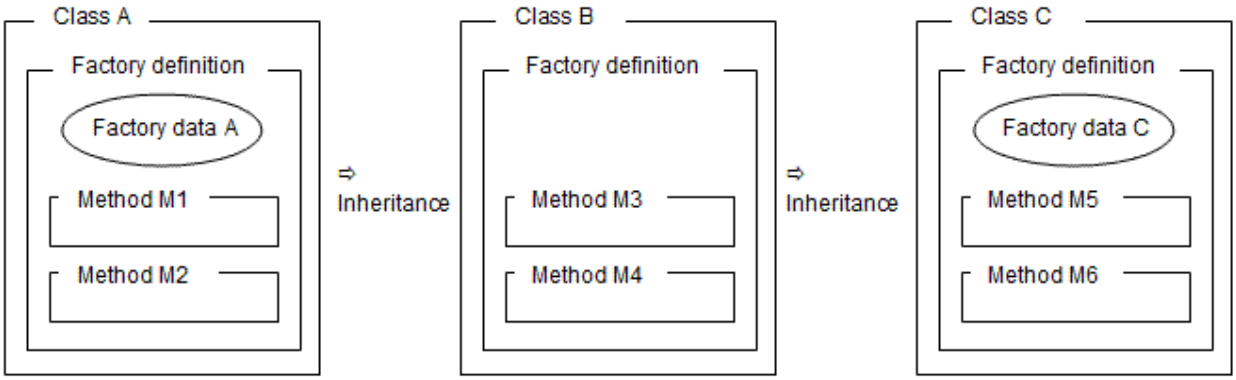
Explanation of diagram

- [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 "16.9.2 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 [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 [1].

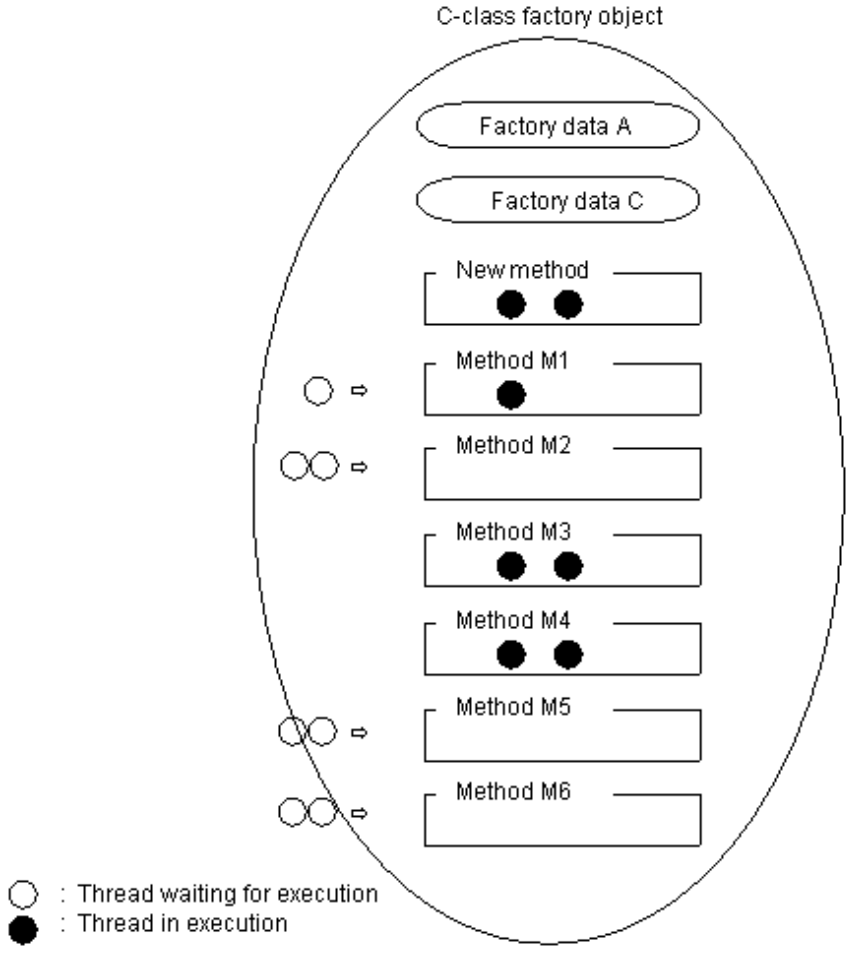
Synchronization Control by the COBOL Runtime System

The thread synchronization control that the COBOL runtime system performs automatically for factory data in the multithread program is described below.

The COBOL runtime system controls threads so that only one thread executes the factory method of a class, for which factory data is defined explicitly, in a factory object. This operation is explained below, using the example given 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 are 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. So, thread synchronization need not be controlled by software for processing that is completed by calling a method once.

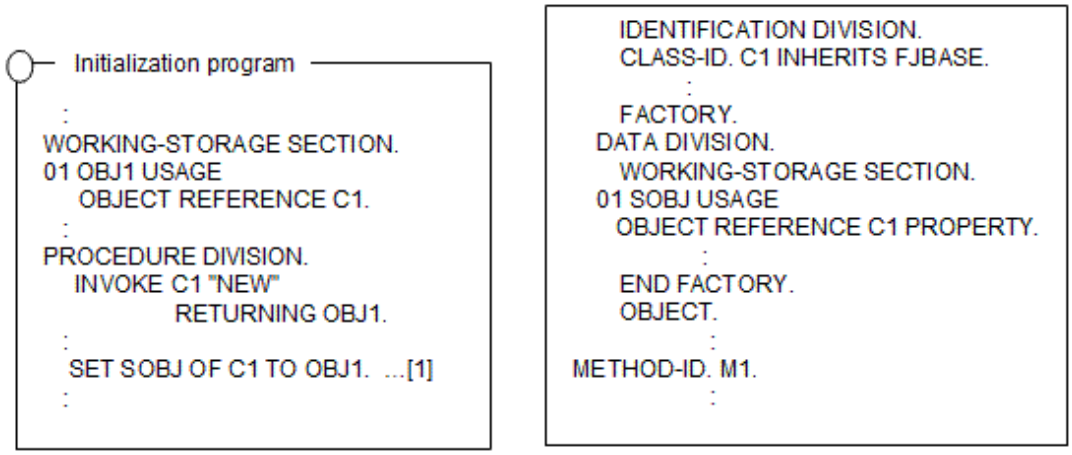
Information

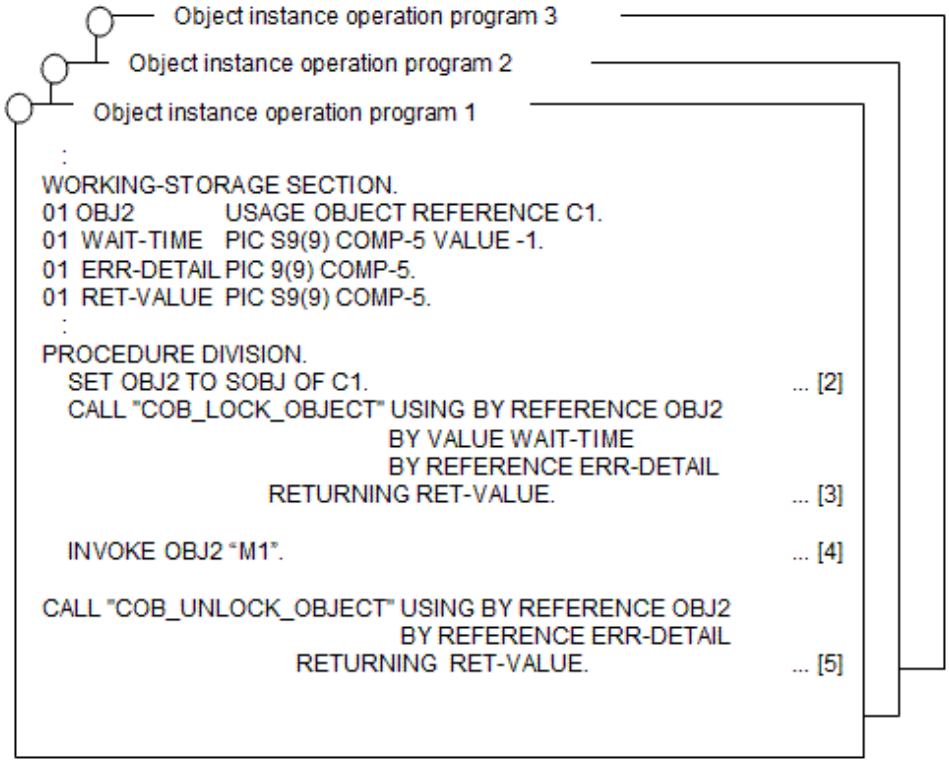
The above synchronization control is performed for each factory object. Therefore, synchronization control of factory objects of the local class is not affected by any method executed in a factory object of an inherited class.

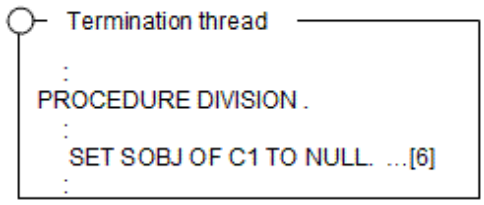
16.4.3.3 Object Instance

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.







Explanation of diagram

- [1] In the initialization program, the property method of the C1-class factory object is called, and the C1-class object instance is set in factory data. The initialization program is activated first in the execution environment.
- [2] The property method of the C1-class factory object is called, and the C1-class object instance set in [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 defined, or the object data is referenced only. Refer to "16.9.2 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 [3].
- [5] The lock of the object instance is released. Releasing the lock enables another thread to obtain the lock by using the method described in [3].
- [6] The termination thread is called last in the execution environment only once. In this thread, the property method of the C1-class factory object is called to delete the object instance set in the factory data.

16.5 Basic Use

Only if the program has neither data nor files in an Object-Oriented factory object, can a process model program be transferred to multithread by recompiling and re-linking it by specifying the multithread mode option.

For mode details, refer to "16.3.2 Data Treatment of Multithread programs" and "16.7.1 Compilation and Link".

Be sure to read "16.4 Resource Sharing Between Threads" if the factory object contains factory data and files.

The section below explains the function that should be noted when a process model program is compiled to a multithread program.

16.5.1 Dynamic Program Structures

This section explains notes for the case where the CANCEL statement is executed for a subprogram called by the dynamic program structure:

- Even a multithread program allows the program to be initialized with the CANCEL statement. However, a common object program of the program specified with the CANCEL statement is not deleted from the virtual memory. As a result, the program executes normally even if other threads execute the CANCEL statement for the program under execution.

16.5.2 Use of the Input-Output Module

This section describes how to develop a multithread program that operates files by the input-output module.

16.5.2.1 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 by operating separate file connectors among threads, regardless of the internal and external attributes.

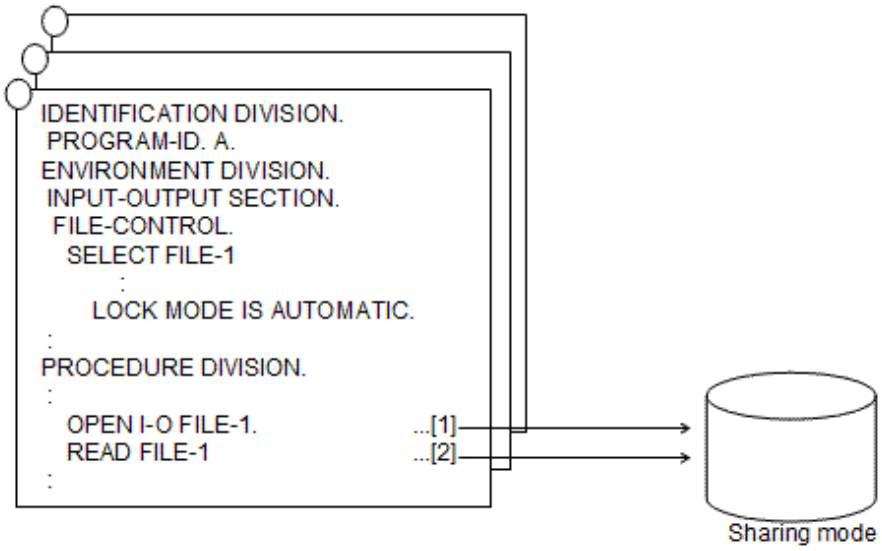
How the same file is shared by operating internal attribute file connectors is described below.

Files Defined in Program/Method/Object

A file can be shared among threads by allocating it to files defined in file description entry a program, a method, 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 operating the object reference identifier of each object instance.

An example of the sharing of a file defined in a program is shown below:



Explanation of diagram

- [1] The file is opened in sharing mode.
- [2] A record is read.

Similarly to a process model program, to operate the same file, perform processing in accordance with exclusive control of files. For information on the exclusive control of files, see "6.7.3 Locking Control of Files".

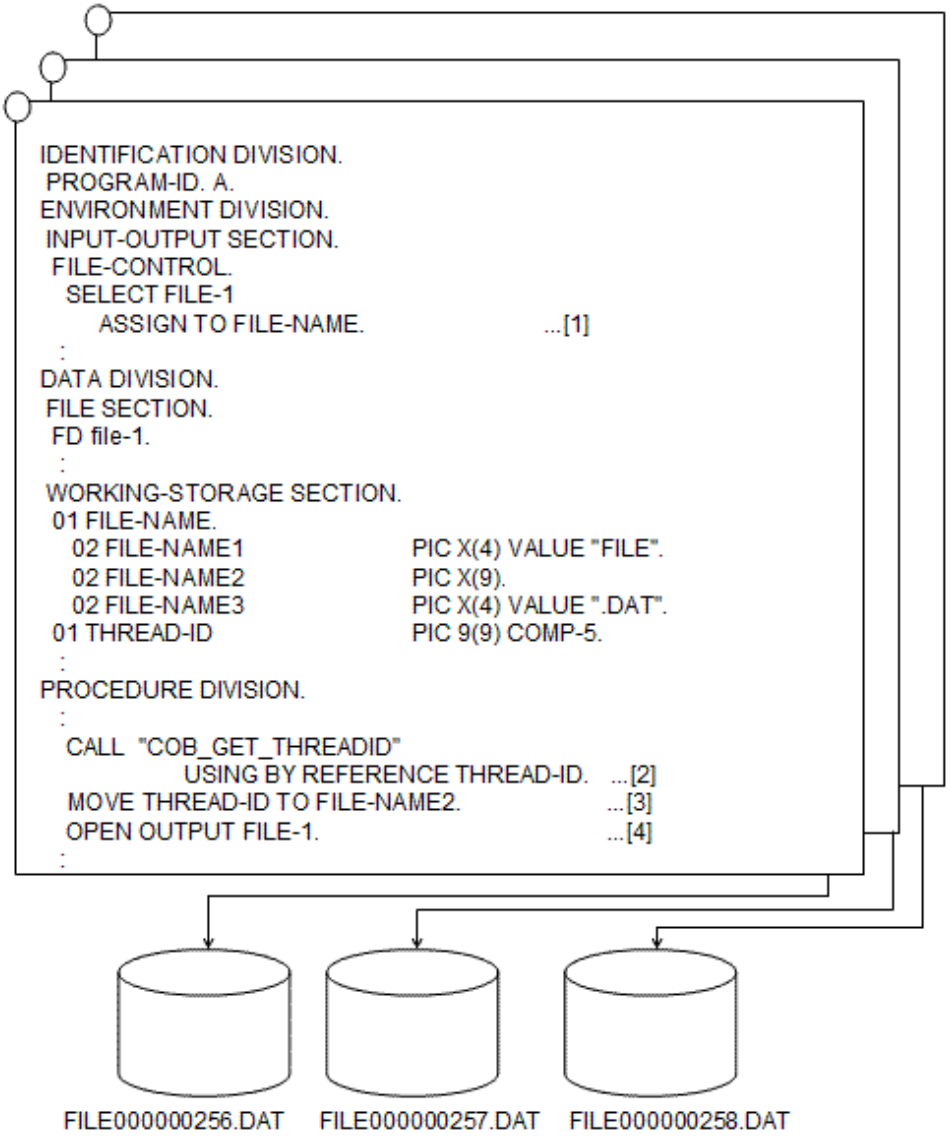
Note

A file defined in a file description entry in a factory object is shared among threads. For information on this type of file, see "16.6 Advanced Use".

16.5.2.2 Operation of a Multiple Files by Executing the Same Program

To operate the same program as a multithread program, the same file is operated among threads.

Below is described how a separate file is operated in each thread by executing the same program:



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 described above, the thread ID obtained by the thread obtain subroutine is set as the file-name, thus enabling a separate file to be operated in each thread by executing the same program. For information on the thread obtain subroutine, refer to "G.1.2 The 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.

16.5.2.3 Note

Operating multiple multithread model programs in which a single OPEN statement specifies multiple files may cause a deadlock, depending on the order in which the files are specified. To operate multiple multithread model programs, specify the OPEN statement in each file or specify file names in the same sequence in each OPEN statement.

16.5.3 Using Print Function

This section explains notes for the case where print files or presentation files are shared by operating the same file connector among threads.

When executing such a processing as executing more than one input-output statement generates forms, the order of executing the input-output statements is not synchronized if input-output statements from more than one thread to the same file connector are executed. Therefore, the order of outputting the print data is not constant. As a result, unintended print result may be obtained.

To obtain intended print results, the race with the input-output statement of other threads should be controlled through series of processing for the same forms.

To prevent the race condition with other threads, synchronization control of threads should be executed before and after a series of processing.

For more information about synchronization control of the file and thread with the same file connector between threads, refer to "16.5.2 Use of the Input-Output Module".

16.5.4 Using the ACCEPT and DISPLAY Statements

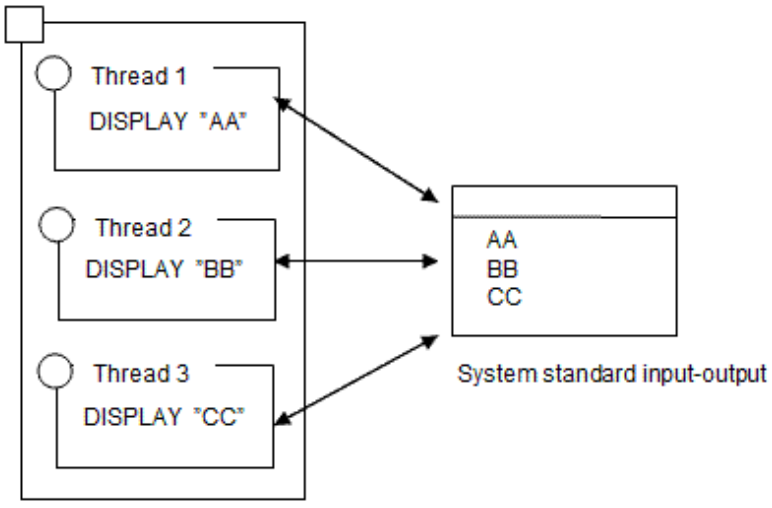
This section describes how to use the ACCEPT and DISPLAY statements in multithread environments.

16.5.4.1 ACCEPT/DISPLAY Statements

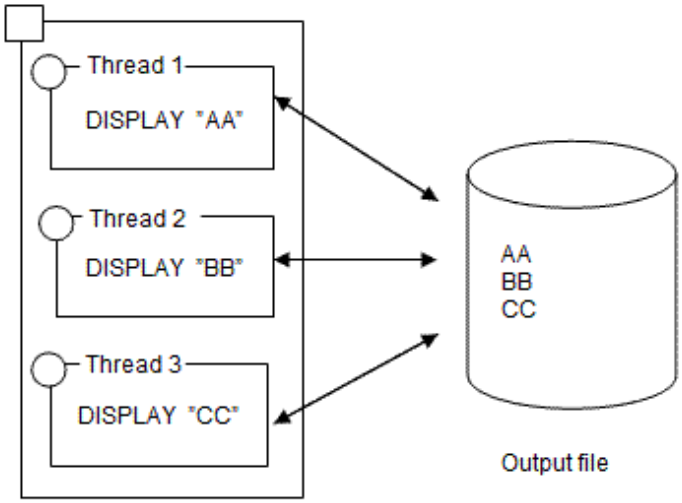
In multithread mode, for ACCEPT/DISPLAY function using system standard input and output and files, one standard input and output and one file are shared among processes.

Example: Input-output example

- When system standard input and output is used:



- When a file is used:



Synchronization of data input-output data is controlled for each ACCEPT or DISPLAY statements if the ACCEPT/DISPLAY function is performed.

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.

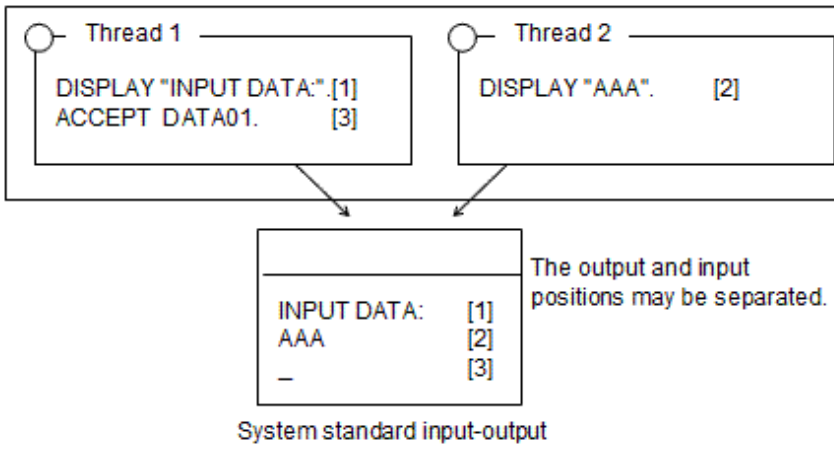
To control synchronization of the execution order, for example, to execute an ACCEPT statement immediately after a DISPLAY statement, use the thread synchronization control subroutine.

Refer to "16.9 Thread Synchronization Control Subroutine".

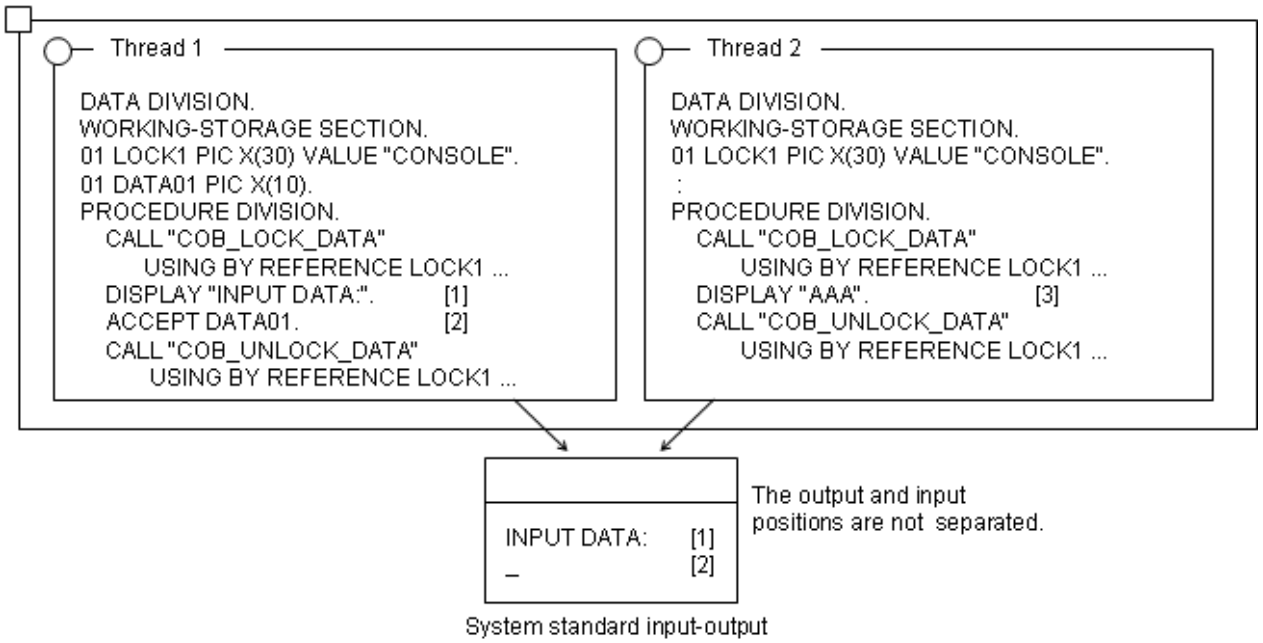
Example

Synchronization control of two or more ACCEPTS and DISPLAY statements ([1] to [3]: Execution order)

- When the thread synchronization control subroutine is not used:



- When the thread synchronization control subroutine is used:



Which is to be used, a standard input-output 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.

If a redirection specification causes the system standard output and standard error output to have the same file as their output destinations in a multithread environment, the output file contents are not guaranteed. To output the execution time messages output by the runtime system or the results of the DISPLAY statement associated with the SYSERR function name to any file, specify the output file in the CBR_MESSOUTFILE environment variable.

16.5.4.2 Command Line Argument and Environment Variable Operation Function

16.5.4.2.1 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 operate 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.

16.5.4.2.2 Environment Variable Operation Function

Environment variable names to be used by the environment variable operation function 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 program, other threads are 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.

16.5.5 Using Object-oriented Programming Function

In process model programs, the timing for ending execution units and exiting execution environment was the same. Therefore, the runtime environment was closed when run unit was ended even if the run unit was not ended after the NULL object was posted in the object reference data item. As a result, the remaining object instance was released from the memory and no problems occurred.

On the other hand, in multithread programs, the timing for ending run units and closing runtime environment is different. Therefore, when the run unit is ended without discarding the object, the object instance will remain in the memory. This causes memory shortage. Therefore, never fail to end the run unit after posting the NULL object in the object reference data item to discard the object. For information about the run unit and runtime environment of the multithread program, refer to "16.3.1 Runtime Environment and Run Unit".

16.5.6 Using the Linkage Function

This section explains notes for the case where a related product is used in the multithread model.

Before you use related products not explained here, confirm the correspondence situation of the related products.

16.5.6.1 Symfoware Linkage

You can use a precompiler to create a multithread program that accesses Symfoware RDB. For details on how to use the precompiler, see the "Symfoware Server RDB User's Guide: Application Program Development".

16.5.6.1.1 Program Description

Describe the embedded SQL statement and create the COBOL program to be accessed in the database. There are no unique multithreads that need to be described. However, you will have to use a SQL extension interface (for creating, destroying, starting and exiting the session) to create programs that are aware of the session. For details about SQL interface, refer to the "Symfoware Server RDB User's Guide: Application Program Development" or the "Symfoware Server SQL Reference Guide".

16.5.6.1.2 Program Compile and Link

To compile and link using the sqlcobol shell procedure, specify the -T option.

To execute pre-compiling, compiling, and linkage separately, follow the procedure below.

- Specify -c in the sqlcobol COBOL option.
- For details about compiling and linkage, refer to "16.7 Method from Compilation to Execution". Also specify option -l sqldrvm and -L /opt/FSUNrdb2b/lib at the time of linkage.

16.5.6.1.3 Program Execution

To make it possible for a Symfoware link 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)" in "16.7.2.2.1 Specification Format of Execution Environment Variables".

16.5.7 Calling Programs that Cannot Execute Multiple Operation

When a following program is called, a synchronization control by the data lock subroutine is needed. For information about the data lock subroutine, refer to "16.9.1 Data Lock Subroutines".

- Though it is a multithread mode, it guarantees the operation only in the environment in which more than one thread does not operate at the same time. (Programs offered from a related product are included.)

16.6 Advanced Use

16.6.1 Using the Input-Output Module

The basic use method explained the sharing of the file that operated a different file connector between threads.

This section 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.

When a single file is accessed by the same file connector, the status of the file position indicator is changed by executing the input-output statement. Therefore, to perform operation in multithread mode, a program must be designed considering the status of the file position indicator.

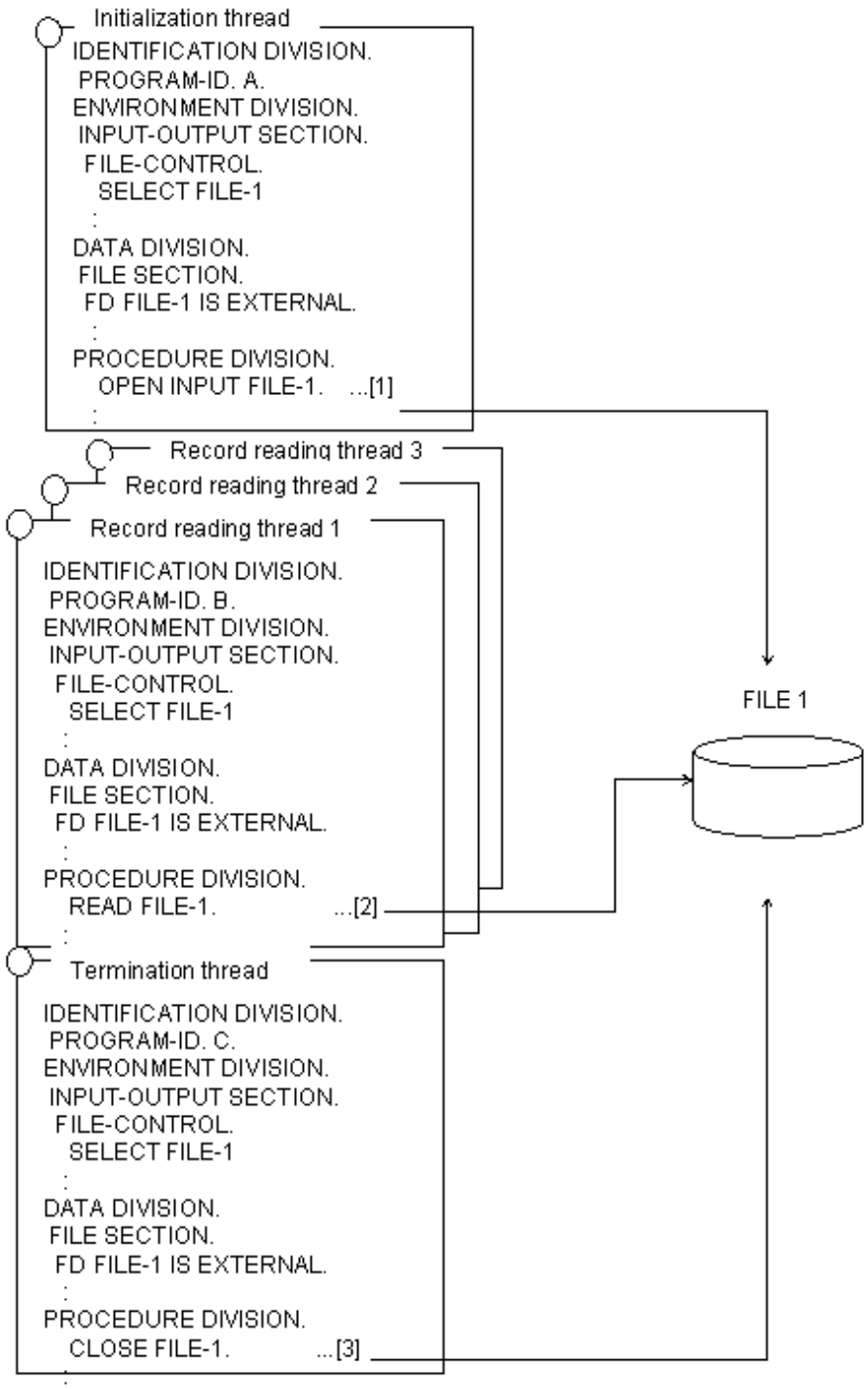
16.6.1.1 External File Shared between Threads

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 when compiling the multithread program.

The EXTERNAL clause can also be specified in the file description entry in the method.

The following is an example of the program using the external file shared between threads:



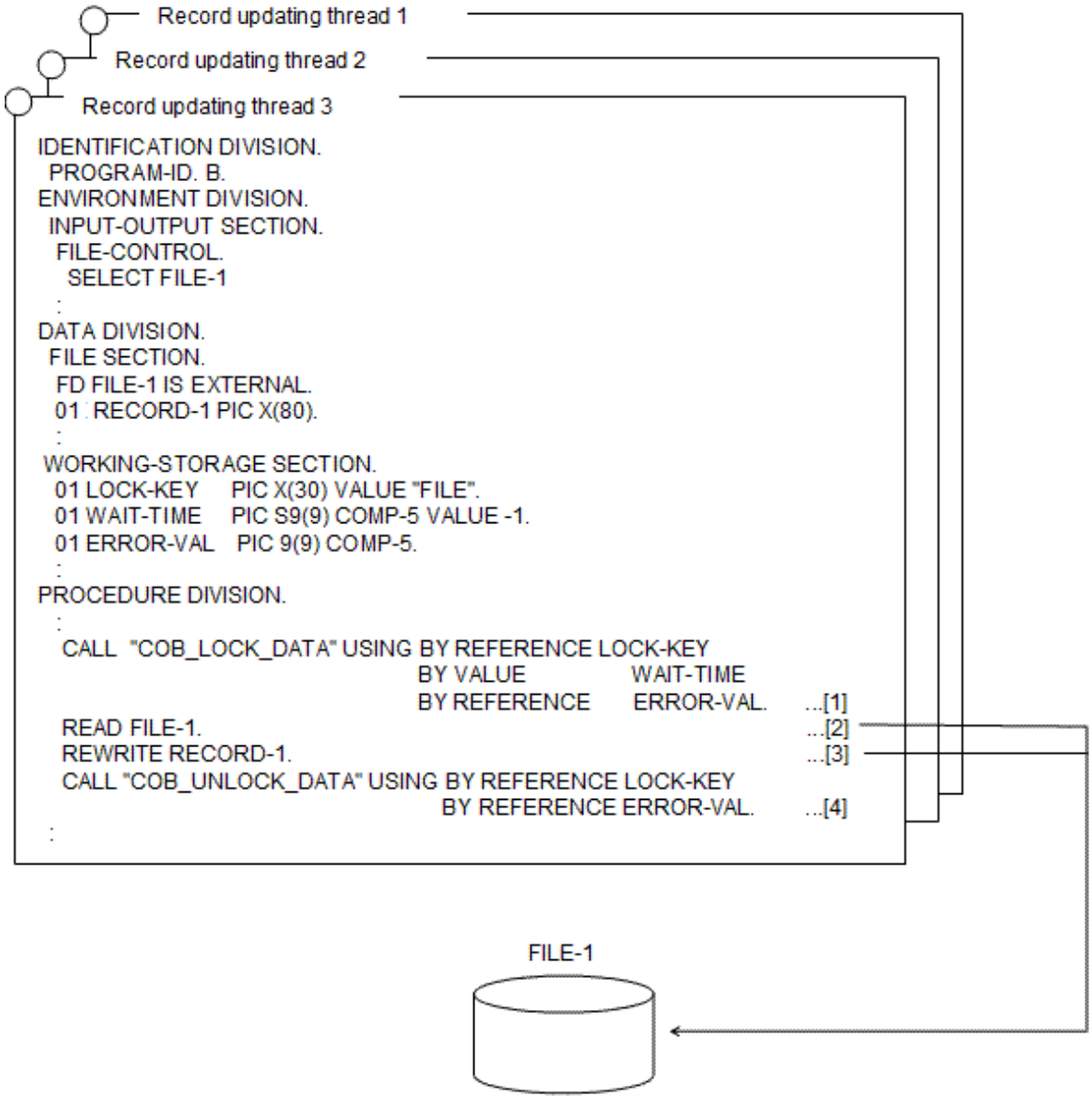
Explanation of diagram

- [1] The initialization thread is first 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 initialization thread.
- [3] The termination thread is activated last, and only once in the runtime environment. The end thread closes the file opened by the initialization 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 is used. Using this subroutine, for example, prevents another READ statement from being executed by another thread between the READ statement and the REWRITE statement.

The following is an example of the thread synchronization control program for two or more input-output statements:



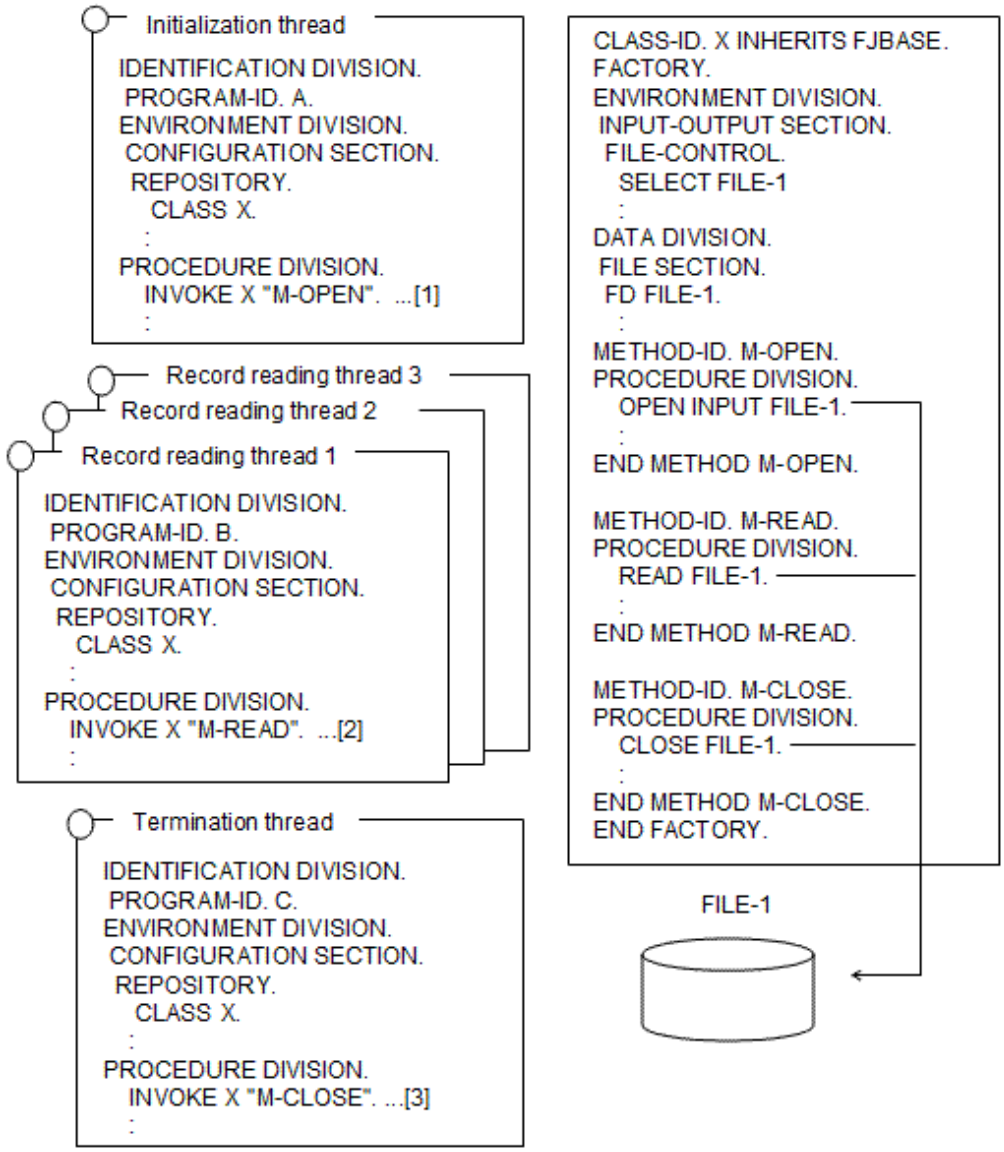
Explanation of diagram

- [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 step 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 "16.9.1 Data Lock Subroutines".

16.6.1.2 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:

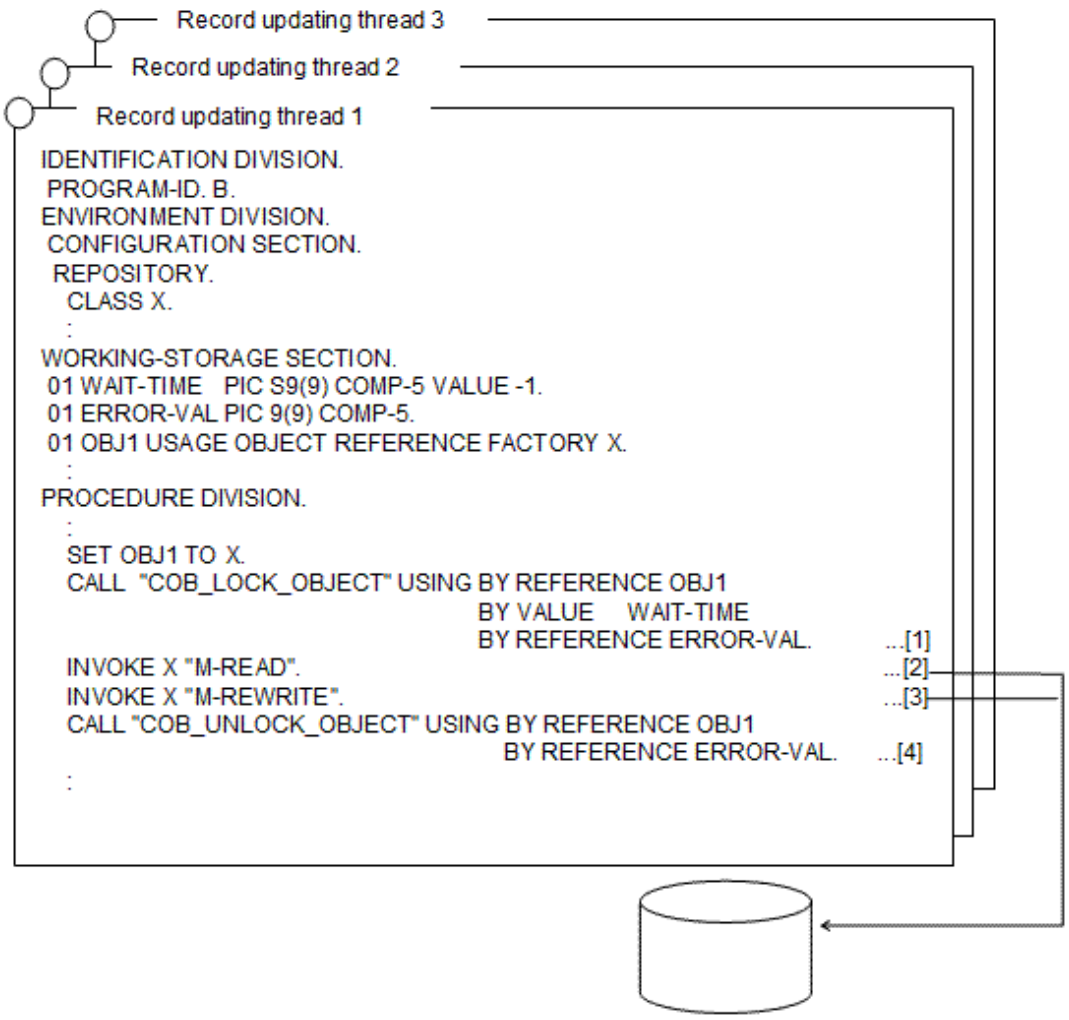


Explanation of diagram

- [1] The initialization thread is first 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 initialization thread.
- [3] The termination thread is lastly activated only once in the execution environment. It calls the factory method M-CLOSE and closes the file opened by the initialization 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 is an example of the thread synchronization control program for several calls of the factory method:



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 step 2.
- [4] The object lock subroutine unlocks the factory object.

For information on the object lock subroutine, refer to "16.9.2 Object Lock Subroutines".

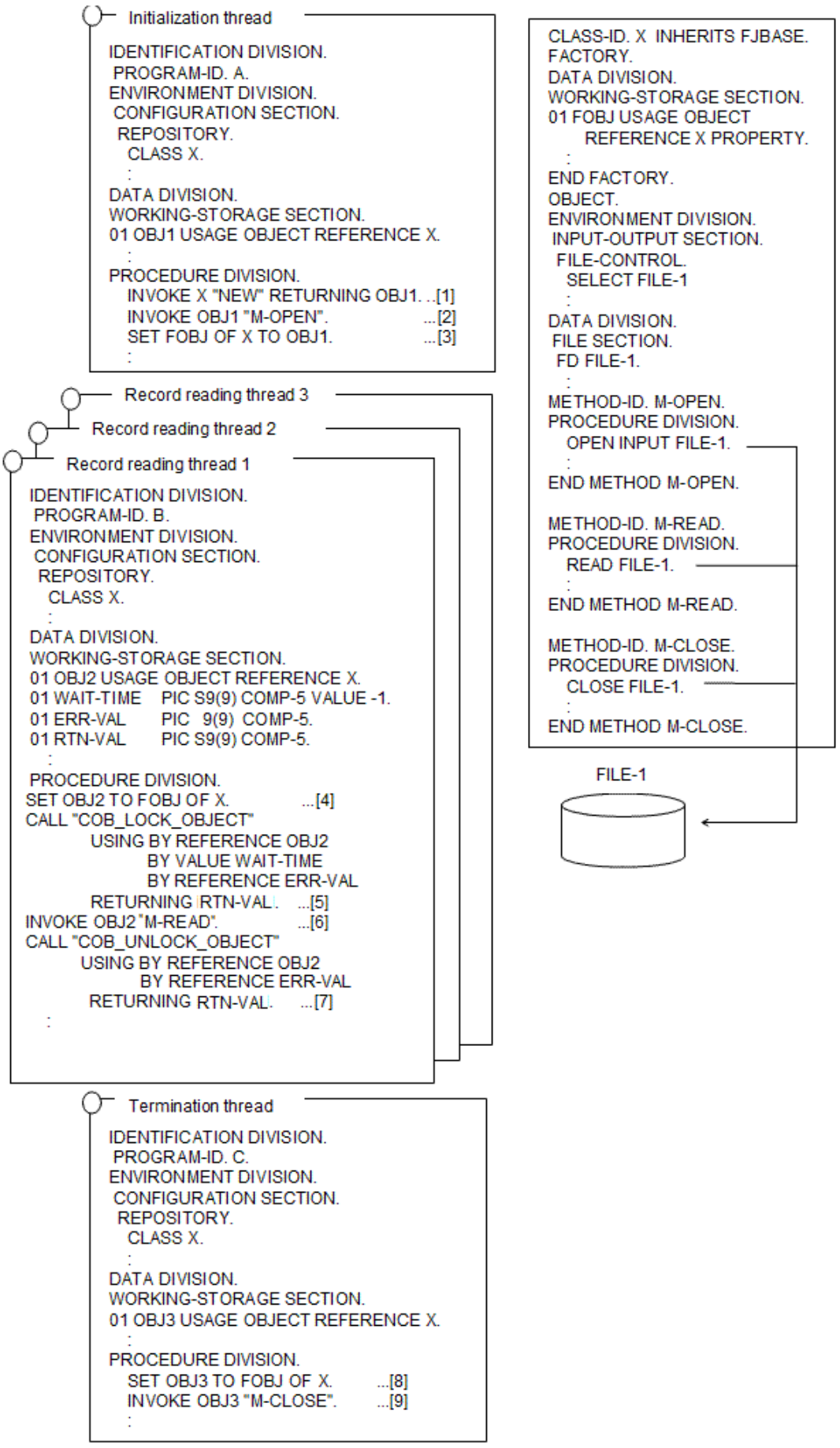
16.6.1.3 File Defined in the Object

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:


Initialization thread
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]
:
Record reading thread 3
Record reading thread 2
Record reading thread 1
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]
:
Termination thread
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.
FILE-1

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 "16.9.2 Object Lock Subroutines".

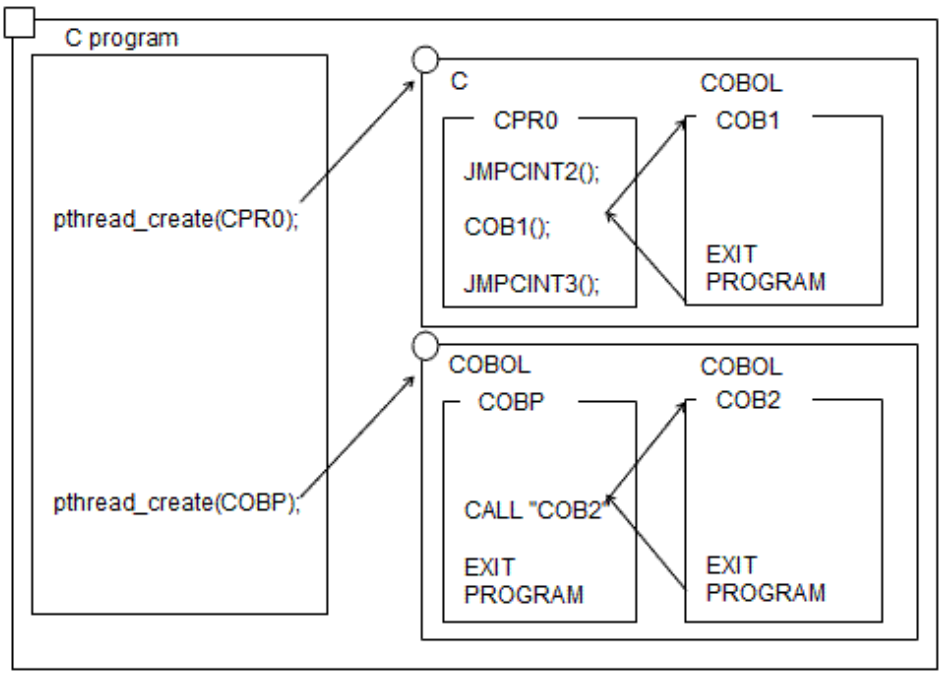
16.6.2 Activating the COBOL Program from the C Program as a Thread

This section explains how to activate the COBOL program from the C program as a thread. POSIX thread is used as an example of the method of starting a thread. For more information about the POSIX thread and other threads, refer to the C language manual.

16.6.2.1 Overview

Unlike cases where COBOL is called from the C program, use the system 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 the 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.



16.6.2.2 Activation

To activate the COBOL program from the C program as a thread, use the system function pthread_create(). The C program that activated the thread executes the next processing without waiting for the end of the thread. Accordingly, the C program may terminate before the COBOL program is activated as the thread ends.

To wait for the end of the thread activated by the C program, use the system function pthread_join().

16.6.2.3 Passing Parameters

To pass an argument to a COBOL program activated from the C program as a thread, specify an actual argument in the fourth argument of system function pthread_create(). Only one actual argument can be specified. The actual argument value that can be passed to the COBOL program activated by the C program as a thread 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 or the USING phase of the ENTRY statement.

16.6.2.4 Return Code (Function Value)

Use the system function pthread_join() to fetch a value specified in the item of the RETURNING phase of the procedure division header 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 "8.3.3 Correspondence of COBOL and C Data Types".

- Function C

```
extern long int *COB(void *arg); ...[1]

                :
main(  ...  )
 {
/*  Thread ID                                       */
  pthread_t  cobtid;
/*  Parameter to be passed to the COBOL program    */
  int    cobprm;
/*  Return code                                     */
  int    cobrcd;                ...[2]
  int    ret;
                :
/*  The COBOL program (COB) is activated as a thread.   */
  ret = pthread_create(&cobtid,
                       0,
                       (void *)COB,
                       &cobprm);
                :
 pthread_join(cobtid, (void **)&cobrcd);
 }
```

- COBOL program(COB)

```
 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.
```

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.

16.6.2.5 Compilation and Link

This section explains the compilation and link, taking the program shown below as an example.

- C program (CPROG.C)

  This program activates COBOL programs COBTHD1 and COBTHD2 as threads, and acquires the return code of each COBOL program.

```
#include <errno.h>
#include <pthread.h>
//              :

/*  The COBOL program to be activated as a thread is declared.    */
extern void *COBTHD1(void *arg);
extern void *COBTHD2(void *arg);

main()
{
/*  Data declaration                                            */
int        cobrcd1;
int        cobrcd2;
//              :

/*  Activate COBTHD1 by specifying 1 in the parameter.          */
cobprm1 = 1;
pthread_create (&cobtid1, NULL, COBTHD1, &cobprm1);

/*  The program waits for the COBTHD1 to end.                   */
pthread_join(cobtid1, (void **)&cobrcd1);

/*  Activate COBTHD2 by specifying 2 in the parameter           */
cobprm2 = 2;
pthread_create (&cobtid2, NULL, COBTHD2, &cobprm2);
//              :
//              :

/*  The program waits for the COBTHD2 to end                     */
pthread_join(cobtid2, (void **)&cobrcd2);
//              :
//              :
}
```

- COBOL program (COBTHD1.cob)

  This program is activated from the C program (CPROG.c) as a thread.

```
 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
*>            :
*>            :
     EXIT PROGRAM.
```

- COBOL program (COBTHD2.cob)

  This program is activated from the C program (CPROG.c) as a thread.

```
 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
*>            :
*>            :
     EXIT PROGRAM.
```

16.6.2.5.1 Compilation

<Compiling the C program>

```
gcc -c CPROG.c
```

<Compiling COBOL program COBTHD1>

```
cobol -c -Tm COBTHD1.cob
```

<Compiling and linking COBOL program COBTHD2>

```
cobol -dy -shared -Tm -o libCOBTHD2.so COBTHD2.cob
```

16.6.2.5.2 Link

<Linking the COBOL program COBTHD1>

```
cobol -dy -shared -Tm -o libCOBTHD1.so COBTHD1.o
```

COBTHD1.o : COBOL program COBTHD1 object

<Linking the C program>

```
cobol -Tm -o CPROG.exe -L. -lCOBTHD1 -lCOBTHD2 CPROG.o
```

CPROG.o : C program CPROG object

16.6.3 Method to Relay Run Unit Data Between Multiple Threads

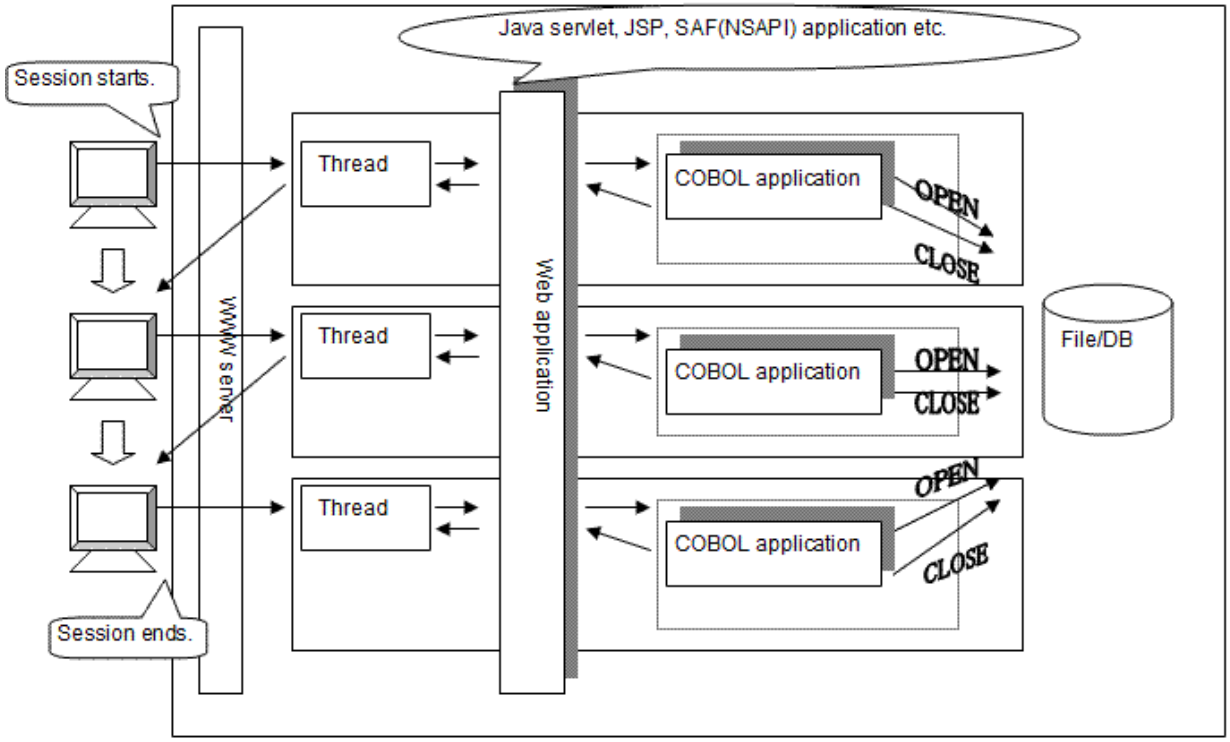
This section describes a method to relay run unit data between multiple threads.

16.6.3.1 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.

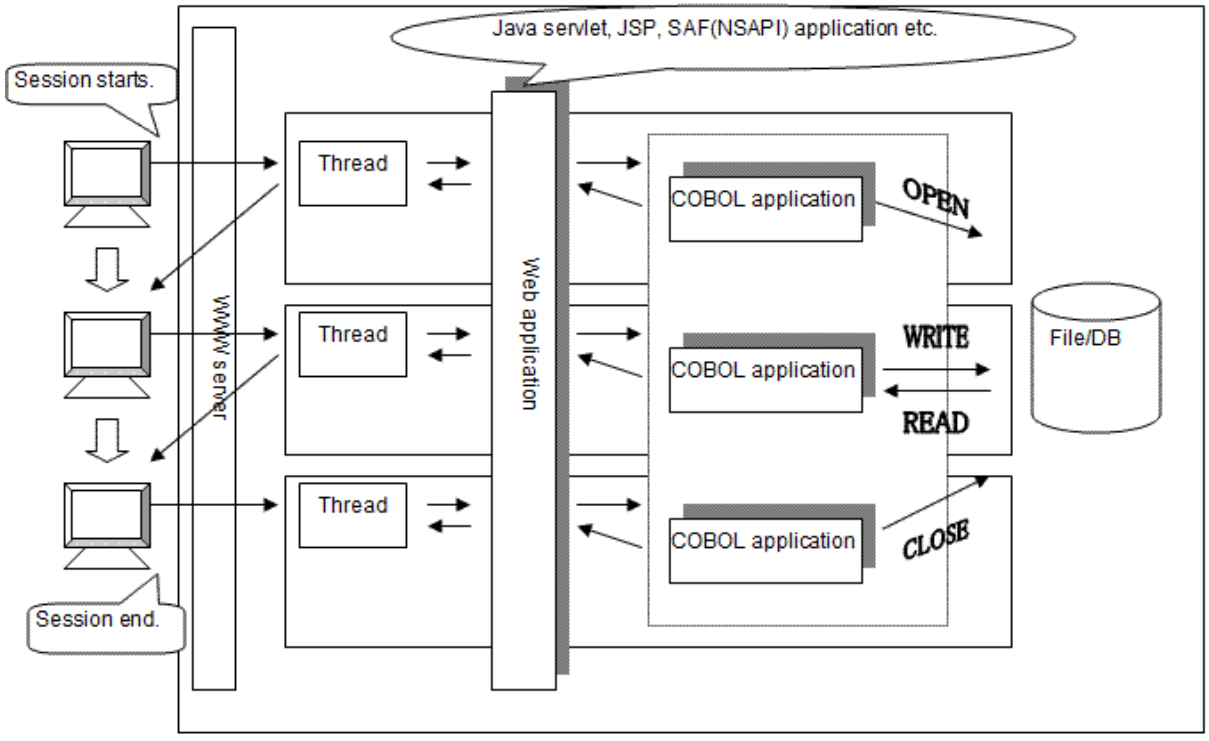
[When the run unit is not inherited]



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.

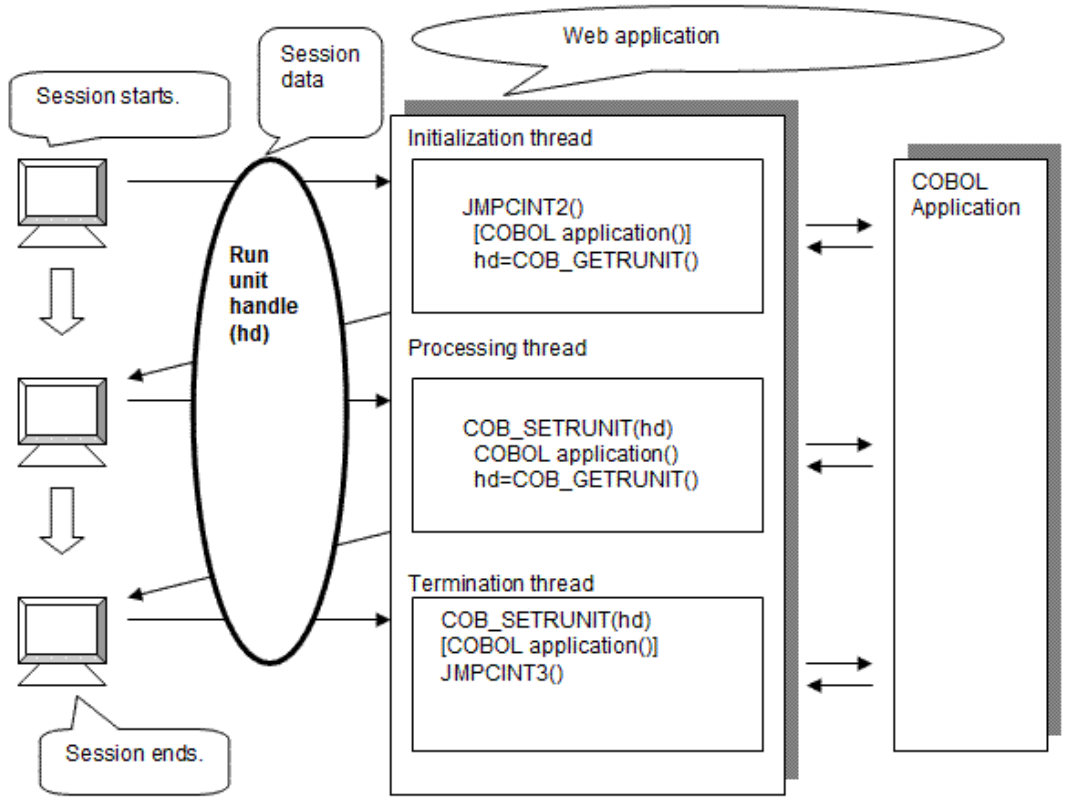
In this case, run unit resources can be relayed between the threads by using two types of subroutines: (1) To return a COBOL run unit handle; and (2) To notify the COBOL run time system of the run unit handle. The linkage between a run unit handle to be relayed and session is made by the caller of a COBOL application, using a management technique such as a cookie.

[When the run unit is inherited]



16.6.3.2 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.

Information

COBOL programs can be easily called from Java programs using J Business Kit (hereafter referred to as JBK). In the JBK wrapping function, the object of Java succeeds the run unit of COBOL programs. Therefore, resources in run units can be took over and used among the COBOL programs called from the object of the same Java.

JBK is a component shipped with Interstage.

16.6.3.3 Use of Subroutine

16.6.3.3.1 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

The specification method describes below.

Description of call (C language calling)

```
Type declaration division :
 extern unsigned long COB_GETRUNIT(void);
```

```
Data declaration division :
 unsigned long hd;  /* COBOL run unit handle storing area */

Procedure division :
 hd = COB_GETRUNIT();
```

Interface

Parameter

none

Return value

Normal: COBOL run unit handle

Error: zero

16.6.3.3.2 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

The specification method describes below.

Description of call (C language calling)

```
Type declaration division :
 extern int COB_SETRUNIT(unsigned long hd);

Data declaration division :
 extern unsigned long hd;
              /* COBOL run unit handle storing area */
              /* (acquired by COB_GETRUNIT)         */
Procedure division :
 COB_SETRUNIT(hd);
```

Interface

Parameter

hd: COBOL run unit handle

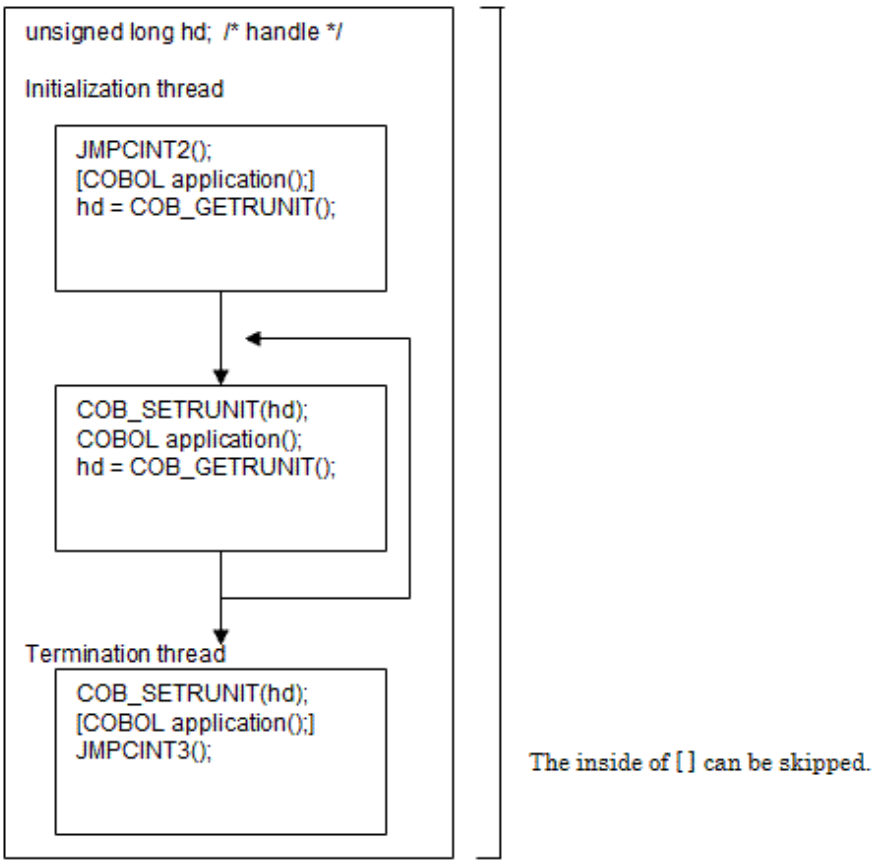
Return code

Normal: zero

Error: The -1 Input parameter is not correct.

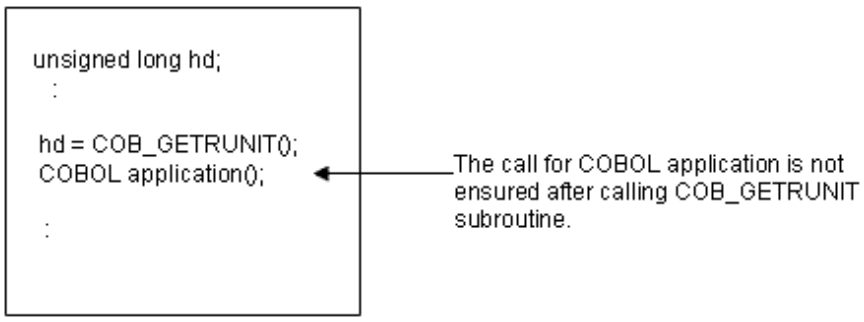
Error: The -2 A COBOL run unit data already exist.

Error: The -99 Other error (contact a system engineer).

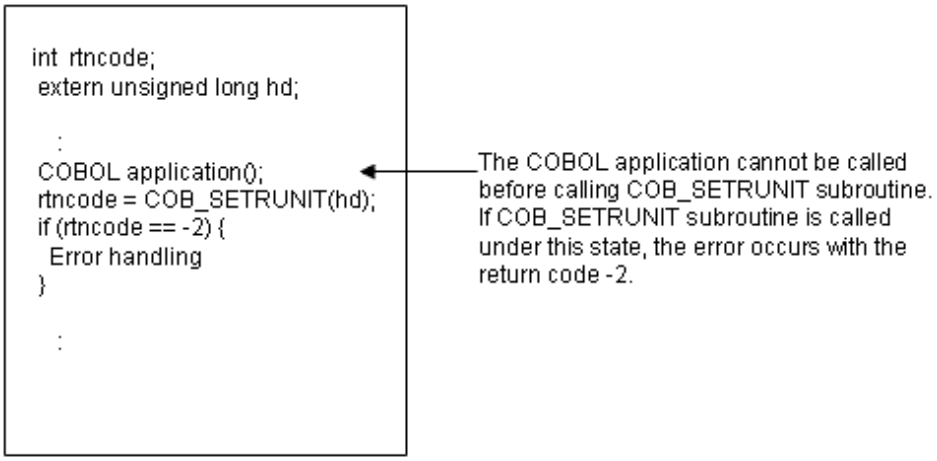


16.6.3.4 Notes

- Where an error occurs, such as a COBOL application run 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.
- 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.



- 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.



- Process identification (PID=) and thread identification (TID=) in COUNT information are process identification and thread identification when the COUNT function outputs information. For information on debugging function for details at the output time, refer to "5.4.2 COUNT Information".
- The restriction on the uses of these subroutines are as follows:
  - Presentation file function.
    The presentation file cannot relay between the threads. Operation must be done in the same thread from an OPEN statement to a CLOSE statement.
  - Print files with the FORMAT phrase.
    The file connector cannot be relayed between the threads. Operation must be done in the same thread from an OPEN statement to a CLOSE statement.

16.7 Method from Compilation to Execution

The method from Compilation to Execution of the multithread program is described below.

16.7.1 Compilation and Link

This section explains how to compile and link the multithread program.

The multithread program differs from the process model program in that it requires the compiler option -Tm option in the compilation procedure. To use the external data shared between threads or external file shared between threads, the compiler option SHREXT is required in addition to -Tm option.

The following table shows the differences between the multithread program and the process model program compilation and linking methods.

Table 16.1 Difference of the method of compilation and linking between the multithread program and the process model program

	Use of the external shared between threads or the external file	Specify with a cobol command when compiling	Specify with a cobol command when linking
Multithread model	Not use	-Tm option or Compiler option THREAD(MULTI)	-Tm option
	Use	-Tm and Compiler option SHREXT	-Tm option

	Use of the external shared between threads or the external file	Specify with a cobol command when compiling	Specify with a cobol command when linking
		or Compiler option THREAD(MULTI) and SHREXT	
Process model	---------	Compiler option THREAD(SINGLE) or None	None

If you specify the -Tm option with cobol command, the multithread model COBOL runtime systems are automatically linked instead of the process model's COBOL runtime system.

For information about the common contents with the process model in the compile and link method, refer to "Chapter 3 Compiling and Linking Programs".

Note

Besides -Tm, THREAD (MULTI) is also effective as a compiler option for creating multithread model. However, it is recommended to use -Tm, which automatically links to COBOL runtime systems for the multithread model by the link processing.

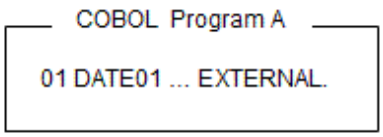
If you executed the following programs, the behavior might not as intended:

- A program that linked an object file compiled by specifying the compiler option THREAD (MULTI) without specifying the -Tm option of the cobol command.
- A program that linked an object file compiled by specifying compiler option THREAD (SINGLE) by specifying the -Tm option of the cobol command.

Reference "16.7.3.2 Program Link Check".

16.7.1.1 Creating Object-Shared Program only with the COBOL Program

The libCOB.so file is created by COBOL program A.

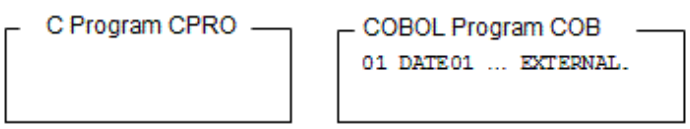



```
cobol -dy -shared -Tm -o libCOB.so A.cob
```

To use the external data shared between threads or external file shared between threads, specify the compiler option SHREXT.

16.7.1.2 Creating Object-Shared Program with the COBOL Program and C Program

The libCCOB.so file is created with C program CPRO and COBOL program COB.



<Compiling C program CPRO>

```
gcc -c CPRO.c
```

Compile the C program so that it runs under a multithread environment.

<Creating libCCOB.so by linking C program CPRO and COBOL program COB>

```
cobol -dy -shared -Tm -o libCCOB.so -WC,SHREXT CPRO.o COB.cob
```

To use the external data shared between threads or external file shared between threads in COBOL program, specify the compiler option SHREXT.

16.7.2 Execution

This section explains the procedure for executing the multithread program. For more information on the process model program, refer to "Chapter 4 Executing Programs".

16.7.2.1 Runtime initialization File

An initialization file for execution is effective in a process. In other words, runtime initialization files for execution shares in multithread programs that run in different threads. Therefore, set the contents of runtime initialization files for execution before executing the programs that operate in the same execution environment.

16.7.2.2 Setting Execution Environment Variables

This section explains the procedure of setting execution environment variables.

16.7.2.2.1 Specification Format of Execution Environment Variables

This section explains environment variables that are valid only for multithread programs.

CBR_SYMFOWARE_THREAD (Specification that Symfoware linkage enables multithread programs to run)

```
CBR_SYMFOWARE_THREAD=MULTI
```

Enables Symfoware linkage multithread programs to run.

For more information, refer to "16.5.6.1 Symfoware Linkage".

CBR_THREAD_TIMEOUT (Specification of wait time of thread synchronization control subroutine)

```
CBR_THREAD_TIMEOUT=wait-time(second)
```

Specify to change wait time in seconds if an infinite wait time is specified in the thread synchronization control subroutine. You can specify from 0 to 32.

For more information, refer to "16.9 Thread Synchronization Control Subroutine".

16.7.3 Check Whether Multithread Model and Process Model are Executing Concurrently

If multithread model and process model are executing at the same time when a COBOL program is created or executed, it may cause a malfunction.

This section explains runtime check for executing multithread programs and process model programs concurrently or a link check needed for creating COBOL programs.

16.7.3.1 Runtime Check

If executing multithread programs and process model programs concurrently in a process, two runtime environments exist and that causes malfunctions. COBOL runtime system detects this and assumes it as an error if a multithread program and process model program are executing at the same time.

16.7.3.2 Program Link Check

COBOL programs do not run as intended or a runtime error occurs if they are executed in the following cases:

- If a multithread program and process model program are running at the same time.
- If an object file for multithread model and object file for process model are linked at the same time.
- If you create a program in the method of linking programs of the process model with the object file compiled for the multithread model.
- If you create a program in the method of linking programs of the multithread model with the object file compiled for the process model.

16.8 Debug Method for the Multithread Program

This section explains how to debug the multithread program.

16.8.1 Debugging the Multithread Program

Problems may occur if a multithread program is executed under the following two conditions:

- If a multithread program is executed in the process model.
- If more than one program is executed at the same time.

If a problem occurs when you execute the process model, it can be debugged with the legacy debugging method.

Problems do not always occur when more than one program is executed at the same time. This is because the order of executing the threads is not decided. Problems that occur when more than one program is executed at the same time generally show a statistical phenomenon tendency. Therefore, when problems at the execution level are detected, debugging which uses trace information is more effective than debugging with the breakpoint is set.

The best debugging method for each of the problems is different. Therefore, it is important to identify which type of the above problems occurred by reproducing in the following method:

- Executing only one multithread program in a process.
- Changing it to process model and executing the program.

Taking the above actions speeds up the process of solving the problem because it narrows down the conditions that could have caused the error.

16.8.2 Debug Function for the Multithread Program

The debug method is generally the same for the following functions provided by COBOL even if the multithread program is used:

- TRACE function.
- CHECK function.
- COUNT function.

These functions can be used as well as the function to debug the process model programs.

This section explains some notes must be observed when debugging the multithread program.

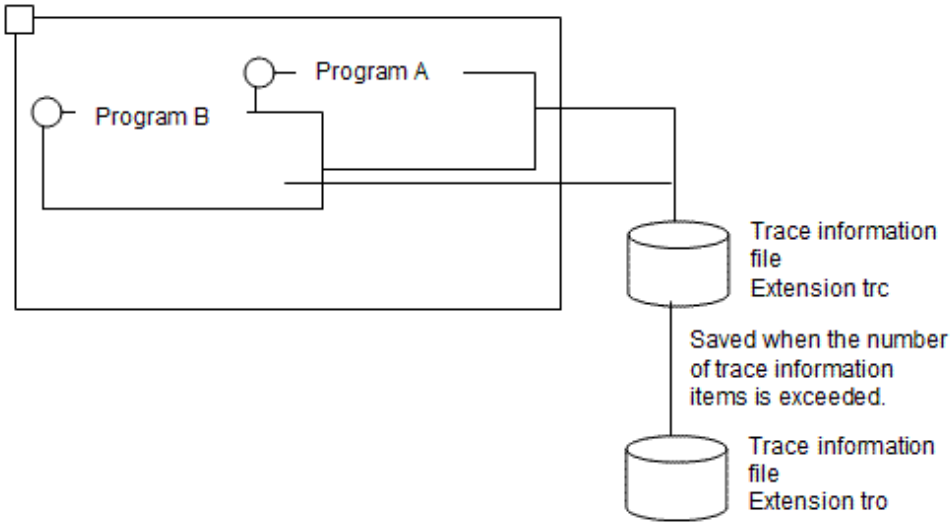
For information on basic usage of debug function, refer to "Chapter 5 Debugging Programs".

16.8.2.1 TRACE Function

The content of the trace information remains unchanged. Refer to "5.2 Using the TRACE Function".

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 2000-03-31  TIME 14:50:54
PID=00003488

TRACE INFORMATION

[1]      1  A           DATE 2000-03-31  TIME 14:50:47  TID=00000004
         2               7.1 TID=00000004
         3               8.1 TID=00000004
         4              11.1 TID=00000004
[2]      5      EXIT-THREAD TID=00000004
[1]      6  B           DATE 2000-03-31  TIME 14:50:48  TID=00000005
         7               9.1 TID=00000005
         8              10.1 TID=00000005
         9              13.1 TID=00000005
[3]'    10              14.1 TID=00000005
[1]     11  C           DATE 2000-03-31  TIME 14:50:49  TID=00000006
        12               7.1 TID=00000006
[3]     13    JMP0015I-U [PID:00003488 TID:00000005] CANNOT CALL PROGRAM 'D'. ld.so.1: PROG: fatal:
D: Can't find symbol.   PGM=B. LINE=14
```

Explanation of diagram

- [1] Thread ID assigned to the program. The thread of program A is 00000004. The thread of program B is 00000005. The thread of program C is 00000006.
- [2] Thread end notification message. The thread ended for program A whose thread ID is 00000004. The following can be obtained from the above results:
  - Program A whose thread ID is 00000004 ran normally.
  - Program B whose thread ID is 00000005 caused an exception in the execution statement in line 14.
  - Program C whose thread ID is 00000006 is forced to terminate.
- [3] Runtime message.

  A runtime error occurred in the program B whose thread ID is 00000005. Number [3] shows that the 14th line was the last line executed by program B.

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 "CBR_SYSERR_EXTEND(Specify the SYSERR output information extension)".

Note

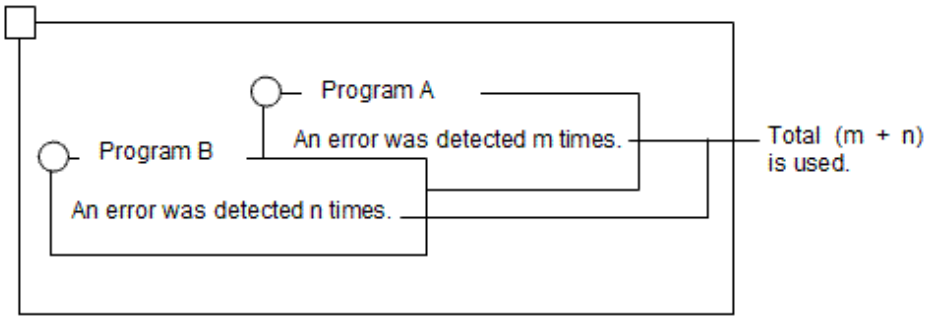
When many threads are executed at one time, the trace information is written in a single file. Adjust the number of trace information items (specification of "r" in environment variable information GOPT) so that much more trace information is output to a single file.

16.8.2.2 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 "5.3 Using the CHECK Function".

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:



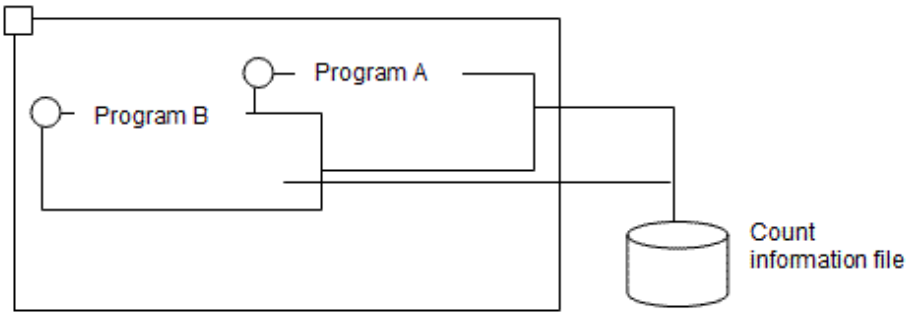
16.8.2.3 COUNT Function

The content of the count information remains unchanged. Refer to "5.4 Using the COUNT Function".

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 2000-03-31  TIME 13:19:43
PID=0000063C  TID=00000004
                                        [1]
  STATEMENT EXECUTINPUT-OUTPUT N COUNT    PROGRAM-NAME : A
```

```
  :

COBOL  COUNT INFORMATION (END OF RUN UNIT)        DATE 2000-03-31  TIME 13:19:43
PID=0000063C  TID=00000005
                                          [1]

  STATEMENT EXECUTION COUNT    PROGRAM-NAME : B

  :
```

Explanation of diagram

- [1] Thread ID assigned to the program. The thread ID of program A is 00000004. The thread ID of program B is 00000005.

16.8.2.4 Interactive Remote Debug Function

The method of operating the remote debug function of NetCOBOL Studio need not be changed when a process model program is executed. For more information, refer to "18.1 How to Use Remote Debug Function of NetCOBOL Studio".

When debugging multithread model, synchronization control is automatically executed so as not to execute more than one COBOL program concurrently.

Therefore, when more than one program is executed at the same time, the same debugging process as process model can be executed without changing it into the process model.

When a remote debug function of NetCOBOL Studio is used, any problems that occur when more than one program is executed at the same time may not be reproduced.

This occurs because the order of executing the threads is changed. Debug such a problem using information output by COBOL trace information along with the method of investigating a statistical phenomenon tendency.

16.8.2.5 How to Identify Trouble Generating Parts

For information on how to identify the location and cause of an abnormal program termination that occurs when a fatal error occurs in the COBOL runtime system, see "5.6 How to Identify the Portion Where an Error Occurred at Abnormal Termination".

16.9 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.

To call a thread synchronous control subroutine using a dynamic program configuration, you must use the following entry information file. For details about entry information, refer to "4.1.3 Subprogram entry information".

```
[ENTRY]
Subroutine-name=librcobol.so
```

16.9.1 Data Lock Subroutines

Subroutine name	Function
COB_LOCK_DATA	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.

The synchronization control performs from a call of the COB_LOCK_DATA to a call of the COB_UNLOCK_DATA.

16.9.1.1 COB_LOCK_DATA

Function

Acquires the lock for the lock key associated with the specified data name.

Calling 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 characters. If the lock key name is less than 30 characters, spaces must be inserted to make it 30 characters long.

- 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) as the value for the 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 system error code is returned.

Return value

- RET-VALUE

  When the operation is successful, 0 is returned. In the process model program, 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 "16.9.3 Error Codes".

16.9.1.2 COB_UNLOCK_DATA

Function

Releases the lock for the lock key associated with the specified data name.

Calling 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 characters. If the lock key name is less than 30 characters, spaces must be inserted at the end to make it 30 characters long.

- ERR-DETAIL

  If a return value is -255, a system error code is returned.

Return value

- RET-VALUE

  When the operation is successful, 0 is returned. In the process model program, 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 "16.9.3 Error Codes".

16.9.2 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.

16.9.2.1 COB_LOCK_OBJECT

Function

Acquires the lock for the specified object.

Calling 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, the wait time can be changed by specifying the wait time(s) as the value for the 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 system error code is returned.

Return value

- RET-VALUE

  When the operation is successful, 0 is returned. In the process model program, 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 "16.9.3 Error Codes".

16.9.2.2 COB_UNLOCK_OBJECT

Function

Releases the lock for the specified object.

Calling 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 system error code is returned.

Return value

- RET-VALUE

  When the operation is successful, 0 is returned. In the process model program, 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 "16.9.3 Error Codes".

16.9.3 Error Codes

This section explains return values for errors generated in the thread synchronization control subroutines.

Error Codes	Meaning and Action	Target Subroutine			
		COB_LOCK_DATA	COB_UNLOCK_DATA	COB_LOCK_OBJECT	COB_UNLOCK_OBJECT
-1	The COBOL execution environment is not open. 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 (COB_LOCK_DATA, COB_UNLOCK_DATA); the specified wait time may be incorrect (COB_LOCK_DATA, COB_LOCK_OBJECT), or the NULL object may be specified (COB_LOCK_OBJECT, COB_UNLOCK_OBJECT) in the object reference.	Notified	Notified	Notified	Notified
-4	The 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. The system error code is set in the ERR-DETAIL parameter.	Notified	Notified	Notified	Notified

Chapter 17 Unicode

This chapter explains how to create COBOL applications that operate using Unicode.

17.1 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 Linux is as follows:

- Shift-JIS

  When expressing Japanese, it is the code commonly used.

  In NetCOBOL, it can be used when the compilation option ENCODE(SJIS) is specified.

  The encoding of the data item and the code systems of all resources regard it as Shift-JIS.

  Shift-JIS can be used as the encoding of the data item in a Unicode application.

- 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.

  Use this if character set used cannot be expressed in neither Native code nor Shift-JIS.

17.2 Overview of Unicode Support

For NetCOBOL, data in different languages can be handled by using Unicode.

17.2.1 Specifying the character encoding

17.2.1.1 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)
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.

17.2.1.2 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.
```

17.2.1.3 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 "ENCODE (data item's encoding 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.

17.2.2 Resources

When Unicode data is processed by NetCOBOL, the translation resource of the source program, etc. can be made with Unicode (UTF-8). The translation resource can be made with Shift-JIS only when using the Shift-JIS as encoding of data.

Note

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. If the maximum length is exceeded, the compiler ignores the part that has exceeded the maximum length. For this reason, note any lines that contain the national language.

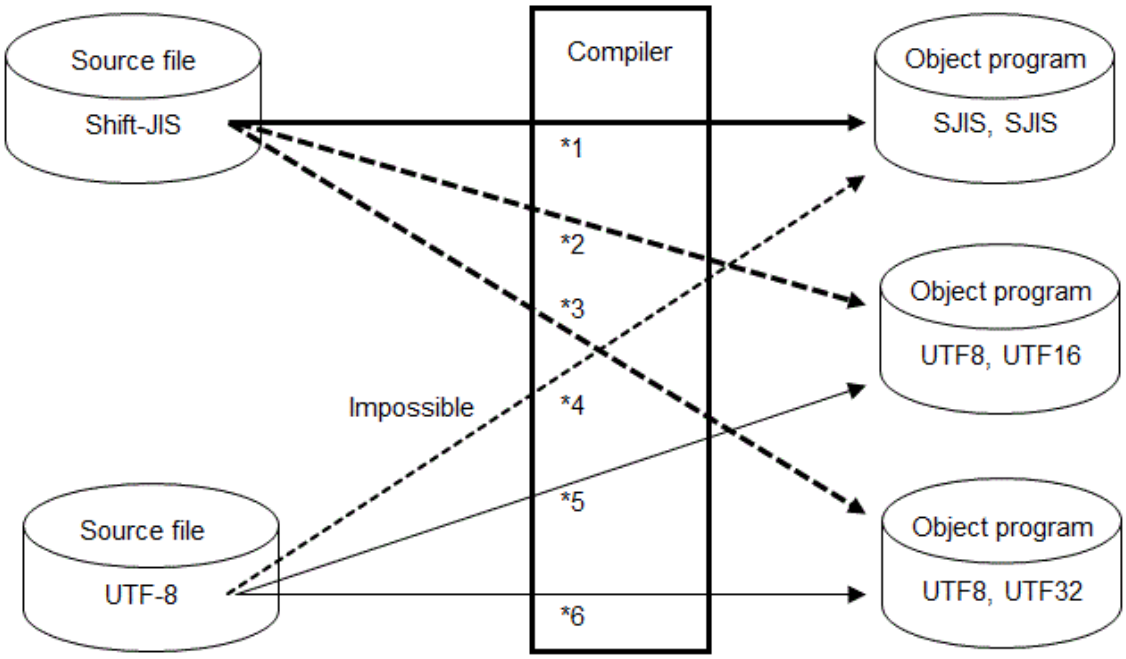
If the following error is output at the time of compilation, it is possible that a line has been left out according to the reference format rules referred to above.

```
cobol: ERROR: system error 'errno=0x016' occurred in 'iconv_error'.
```

In this case, split the lines so that you can use the lines that continue.

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 Unicode.

	ENCODE(UTF8,UTF16)	ENCODE(UTF8,UTF32)	ENCODE(SJIS,SJIS)
SCS(UTF8)	Can combine(default)	Can combine	Cannot combine
SCS(SJIS)	Cannot combine	Cannot combine	Can combine



*1: The compilation options SCS(SJIS),ENCODE(SJIS,SJIS) are required.

*2: The compilation options SCS(SJIS),ENCODE(UTF8,UTF16[,LE|BE]) cannot be specified. A compiler error occurs.

*3: The compilation options SCS(SJIS),ENCODE(UTF8,UTF32[,LE|BE]) cannot be specified. A compiler error occurs.

*4: The compilation options SCS(UTF8),ENCODE(SJIS,SJIS) cannot be specified. A compiler error occurs.

*5: The compilation option is not 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	Shift-JIS or Unicode(UTF-8) (*1)
	COBOL compilation option file(cbi)	IN	ASCII
	Makefile	IN	Unicode(UTF-8)
At Execution	Runtime initialization file Entry information file Class information file Print information file	IN	Unicode(UTF-8)
	Print information file	IN	

	Resource	IN/OUT	Code system
	Window information file	IN	Shift-JIS
	COBOL file (sequential/line sequential/ relative/indexed)	IN/OUT	Shift-JIS or Unicode(UTF-8,UTF-16 or UTF-32) (*2)
	ACCEPT/DISPLAY file (*3)	IN/OUT	Unicode(UTF-8)
	TRACE information file Message Output file COUNT information file	OUT	

*1: The compiler option "A.2.38 SCS(code system of source file)" specifies the code system.

*2: The record definition class defines the code system.

*3: If a BOM is included in the file, input it as part of the data.

Information

BOM

BOM is a Unicode character conventionally used as a marker to indicate that text is encoded in Unicode.

UNIX applications do not use the BOM character. In this product, it is possible to input it even with the UTF-8 file with BOM.

17.2.3 Language Elements

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".

17.3 Notes on Coding

This section provides separate notes and points on how to use Unicode for each language element.

Many of the coding solutions suggested in this section only apply to particular code-types. In other words, these suggestions include coding examples that were originally used to create COBOL applications of high portability (interchangeability), and as such cannot be recommended.

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.
```

17.3.1 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"00000041000000420000043" 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.

17.3.2 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.

17.3.3 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.
```

17.3.4 Intrinsic Function

Intrinsic functions that mutually convert data between UTF-16, UTF-32 and UTF-8 are supported. Use the following intrinsic functions as required:

- To convert UTF-16, UTF-32 data to UTF-8, use the DISPLAY-OF function

- To convert UTF-8 data to UTF-16, UTF-32, use the NATIONAL-OF function

```
WORKING-STORAGE SECTION.
 01 PIC-X      PIC X(14) ENCODING IS U8.
 01 PIC-N      PIC N(7)  ENCODING IS U16L.
*>    :
PROCEDURE DIVISION.
     MOVE FUNCTION NATIONAL-OF(PIC-X) TO PIC-N.
                              *> PIC-N=X"00460075006A0069007400730075"
     MOVE FUNCTION DISPLAY-OF(PIC-N) TO PIC-X.
                              *> PIC-X=X"46756A69747375"
```

Note

- The length of the function result matches that of the conversion result of the character-string specified in the argument. If the value of the argument is not a provided code, the result is not guaranteed.
- The UTF8-OF and UCS2-OF functions were available for use before NetCOBOL V10. DISPLAY-OF and NATIONAL-OF are for use with NetCOBOL V10 and later.

17.3.5 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.

17.3.6 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.

17.3.7 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.
```

17.3.8 Endian conversion function

This function can be used to convert national class endian.

Description format

- Converting big-endian to little-endian

```
CALL "#NATBETOLE" USING [BY REFERENCE] identifier.
```

- Converting little-endian to big-endian

```
CALL "#NATLETOBE" USING [BY REFERENCE] identifier.
```

Functional outline

1. If the name of the program called by the CALL statement is "#NATBETOLE", the function converts *identifier* or all the national items and national edited items in *identifier* from big-endian to little-endian.
2. If the name of the program called by the CALL statement is "#NATLETOBE", the function converts *identifier* or all the national items and national edited items in *identifier* from little-endian to big-endian.
3. If *identifier* is an elementary item that is neither a national item nor a national edited item, the function does not convert it.
4. If *identifier* is a group item and the items subordinate to the group include one of the following items, the function does not convert the group item and subordinate items:
   - An item that is neither a national item nor a national edited item
   - An item where the REDEFINES clause is specified
   - An item where the RENAMES clause is specified

5. The execution results are not guaranteed in the following cases:

   - The program called by the CALL statement specifies an identifier and #NATBETOLE or #NATLETOBE is set in the identifier.
   - The program call is in a format different from the format described above.
   - The name of the program called by the CALL statement is "#NATBETOLE", and the data format of *identifier* or the data format of the national items and national edited items in *identifier* is not big-endian.
   - The name of the program called by the CALL statement is "#NATLETOBE", and the data format of *identifier* or the data format of the national items and national edited items in *identifier* is not little-endian.
   - *identifier* is a group item and the items subordinate to the group include an item that has the OCCURS clause with a DEPENDING phrase.
   - *identifier* is a data item defined in the constant section.
   - *identifier* is a reference modified data item.

17.3.9 ACCEPT/DISPLAY

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.

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 "COBOL files".

17.3.10 COBOL Files

Record sequential files, relative files, indexed files, and line sequential files are input and output without processing their Unicode data. Code conversion of a print file or presentation file (PRT) when the file is output is performed based on the class of each item.

Note the following points:

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.

```
   FILE-CONTROL.
        SELECT OUTFILE ASSIGN TO FILE-NAME.
*>    :
 WORKING-STORAGE SECTION.
```

```
 01 FILE-NAME.              *> Compile error
   02 F-PATH  PIC X(10) ENCODING IS U8.
   02 F-NAME  PIC N(4)  ENCODING IS U16L VALUE NC"file".
*>    :
PROCEDURE DIVISION.
    MOVE "/home/foo/" TO F-PATH.
    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 Files

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.

Indexed 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.

Note

The result is guaranteed only when the ASCII characters are printed.

Form Descriptor (print files with a FORMAT clause)

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.

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

Note

The result is guaranteed only when the ASCII characters are printed.

17.4 Notes on Executing

17.4.1 Files that Output Messages

- Files that output messages at the time of execution

  If the operation mode is Unicode, the code-type of the file (this is specified from the CBR_MESSOUTFILE environment variable information) that outputs the message at the time of execution is as follows:

  - The code system of the existing file is used when appending to an existing file.
  - UTF-8 is used when outputting to a new file.

- Files output by the TRACE function and COUNT function

  When the operation mode is Unicode, the code system of the file output by the TRACE function and the COUNT function is as follows:

  - The code system of the existing file is used when appending to an existing file (COUNT function only).
  - UTF-8 is used when outputting to a new file.

17.4.2 Fonts

If you use the following function, use a font that supports Unicode.

- Print files with a FORMAT clause

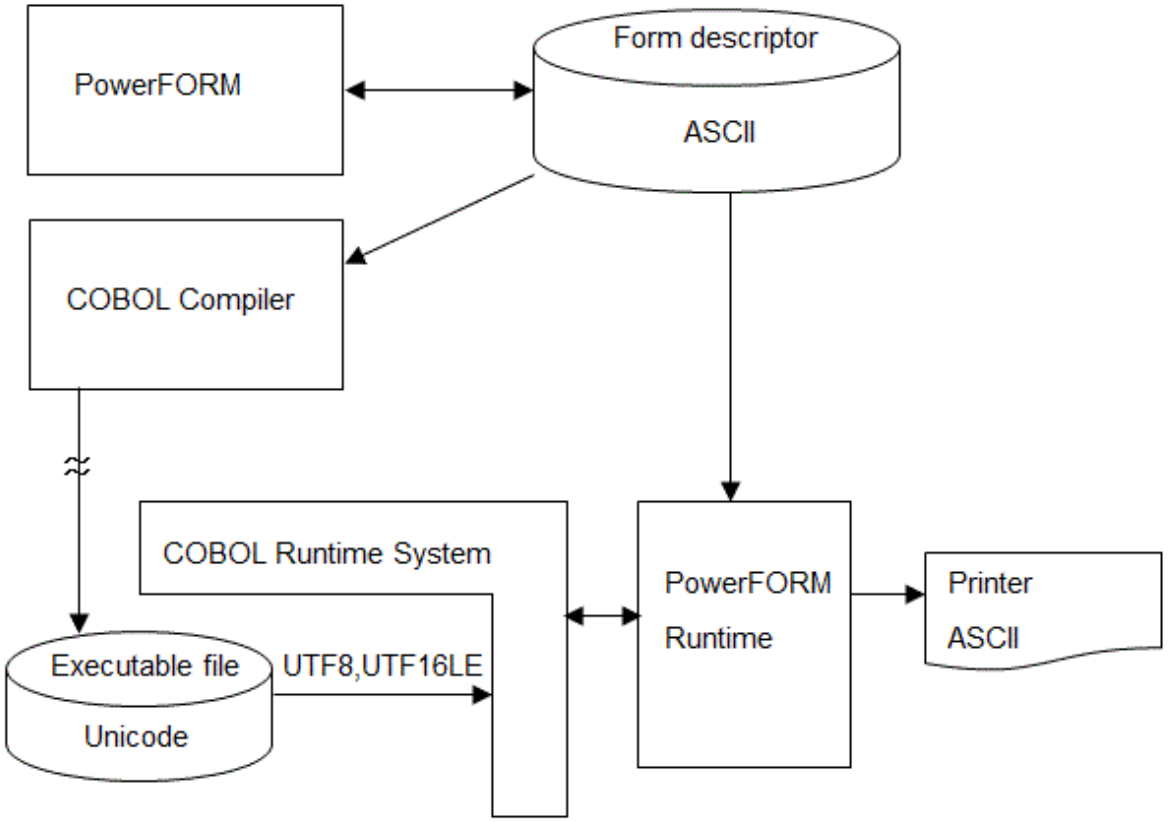
  Specify a font that supports Unicode in the printer information file, and font table. For details, refer to the "PowerFORM Runtime Reference", or "7.1.9 Font Tables".

17.5 Related Product Links

17.5.1 PowerFORM/PowerFORM Runtime

You can use form descriptors created using PowerFORM in Unicode applications. You can also create new definitions using existing ones. PowerFORM Runtime is automatically converted to Unicode at the time of execution.

Only ASCII characters can be used in PowerFORM Runtime. Refer to PowerFORM Runtime manual for details.

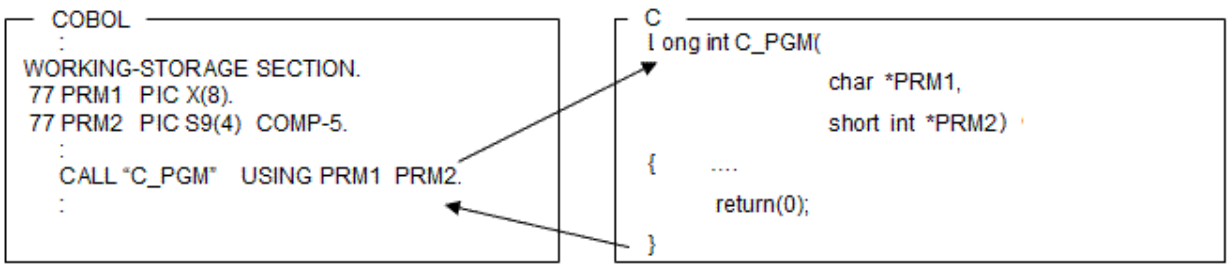


For the operating and functional ranges of a print file with the FORMAT clause and a presentation file (forms printing), refer to "Chapter 7 Printing".

17.5.2 Connection with Other Languages

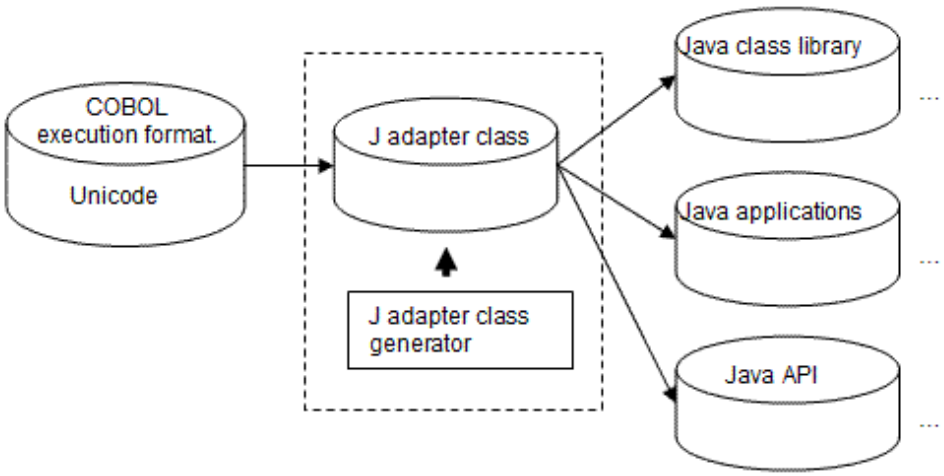
C Language

In C language, you can send and receive data between COBOL alphanumeric character items since the char type is UTF-8 expression. You can also send and receive numerical-type data in the same way as for native code.



Java

The J adapter class can be used via a Java link to directly call a Java class from a COBOL program. In this case, Unicode data can also be sent and received.


COBOL
execution format.
Unicode
J adapter class
J adapter class
generator
Java class library
Java applications
Java API
...
...
...

Chapter 18 Using the Debugger

This chapter describes how to debug created programs using followings.

- Remote debug function of NetCOBOL Studio
- gdb command (the standard debugger of the system)

18.1 How to Use Remote Debug Function of NetCOBOL Studio

Executable programs running under Linux(x64) can be debugged remotely using the remote debug function of NetCOBOL Studio that is included in the Windows edition of NetCOBOL.

This chapter gives an overview of using the remote debug function of NetCOBOL Studio intended for Linux(x64).

The terminology below is used in this chapter.

Remote debug function of NetCOBOL Studio

The remote debug function of NetCOBOL Studio

IP address

Unless otherwise stated, indicates an "IP address or host name"

Server

The computer running the COBOL program

Client

The computer that displays NetCOBOL Studio

Windows edition of NetCOBOL

The NetCOBOL in the NetCOBOL series of products that run under 32-bit Windows and that are sold separately from this product

18.1.1 Overview of the remote debug function of NetCOBOL Studio

The remote debug function of NetCOBOL Studio can be used to debug an executable program running on a different computer in the network.

Debug tasks can be performed using simple operations, such as selecting a button or menu command, from NetCOBOL Studio running under Windows.

The following are the main functions supported by the remote debug function of NetCOBOL Studio :

- Breakpoint
- Unconditional execution
- Single-step execution
- Execution up to a specified line
- Data item reference and changing
- Data item monitoring

To use the remote debug function of NetCOBOL Studio, select the remote build project in the [Dependency] view or the [Structure] view, and then select [Remote development] > [Build in debug mode] from the context menu when NetCOBOL Studio is being used for a remote build.

18.1.1.1 Resource storage locations during remote debug

During remote debug, the resources required for debugging must be stored appropriately, at either the server side, the client side, or at both.

The resources required for debugging are shown in "Table 18.1 Resource storage locations during remote debug".

Table 18.1 Resource storage locations during remote debug

Debug resource	Client side	Server side
Executable program	-	required
Shared object file	-	required
Debugging information file	-	required
Source file	required	-
Library text	required	-
Form descriptor	required	required

18.1.2 Types of remote debugging

The two methods below are available for starting remote debugging.

Method for debugging ordinary programs

From the remote debug function of NetCOBOL Studio, specify the program you want to debug. In this method, the ordinary debugging is used from NetCOBOL Studio remote debug function.

Refer to "18.1.5 Server-side remote debugger connector".

Method for debugging programs that run in a server environment such as Interstage

To debug using this method, the attached debugging is used from NetCOBOL Studio remote debug function.

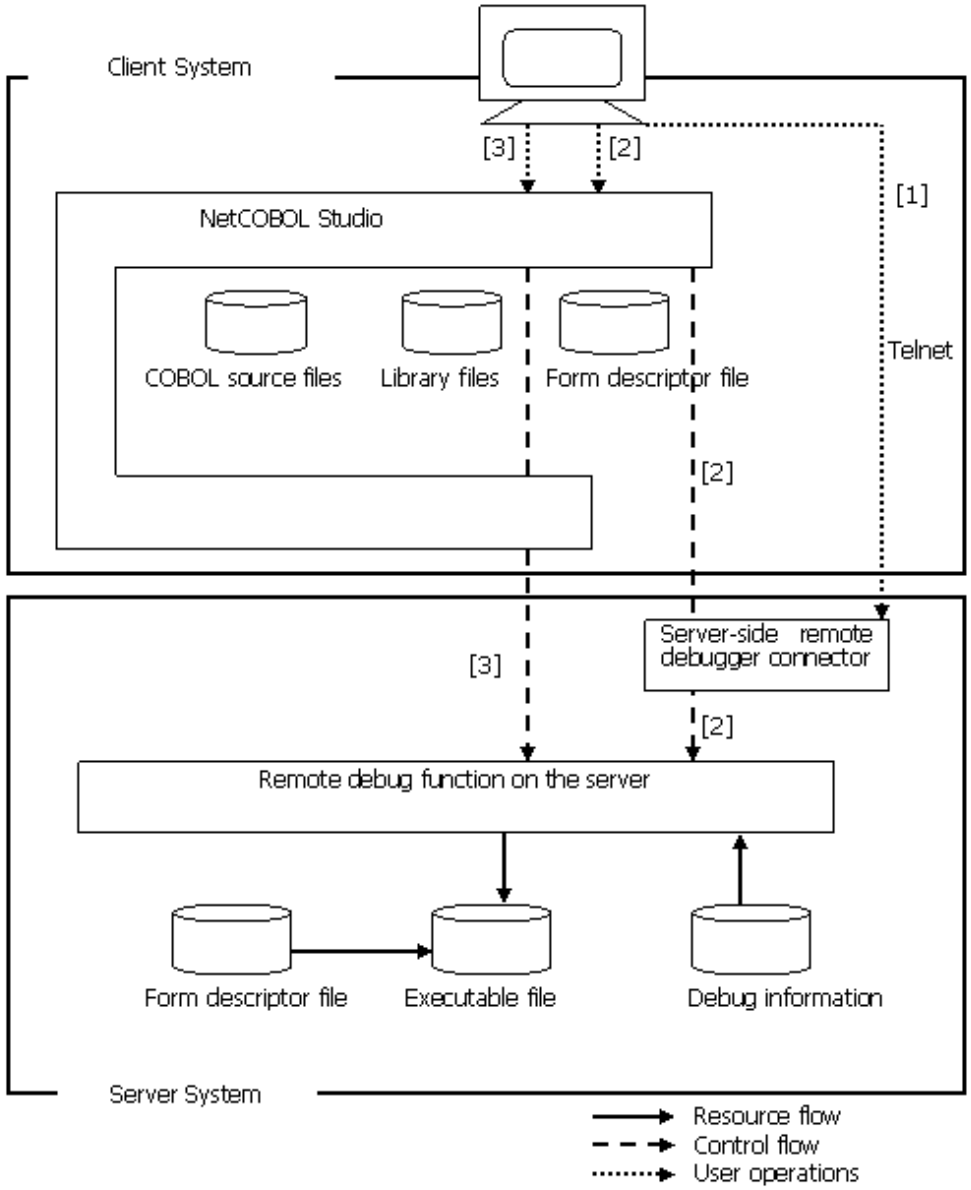
Refer to "18.1.4 CBR_ATTACH_TOOL" and "18.1.6 Client-side remote debugger connector".

Refer to the "NetCOBOL Studio User's Guide" included in the Windows edition of NetCOBOL for NetCOBOL Studio operating methods and remote debug methods.

18.1.3 Debug procedure

This section explains the procedure for using the remote debug function of NetCOBOL Studio to debug a COBOL program.

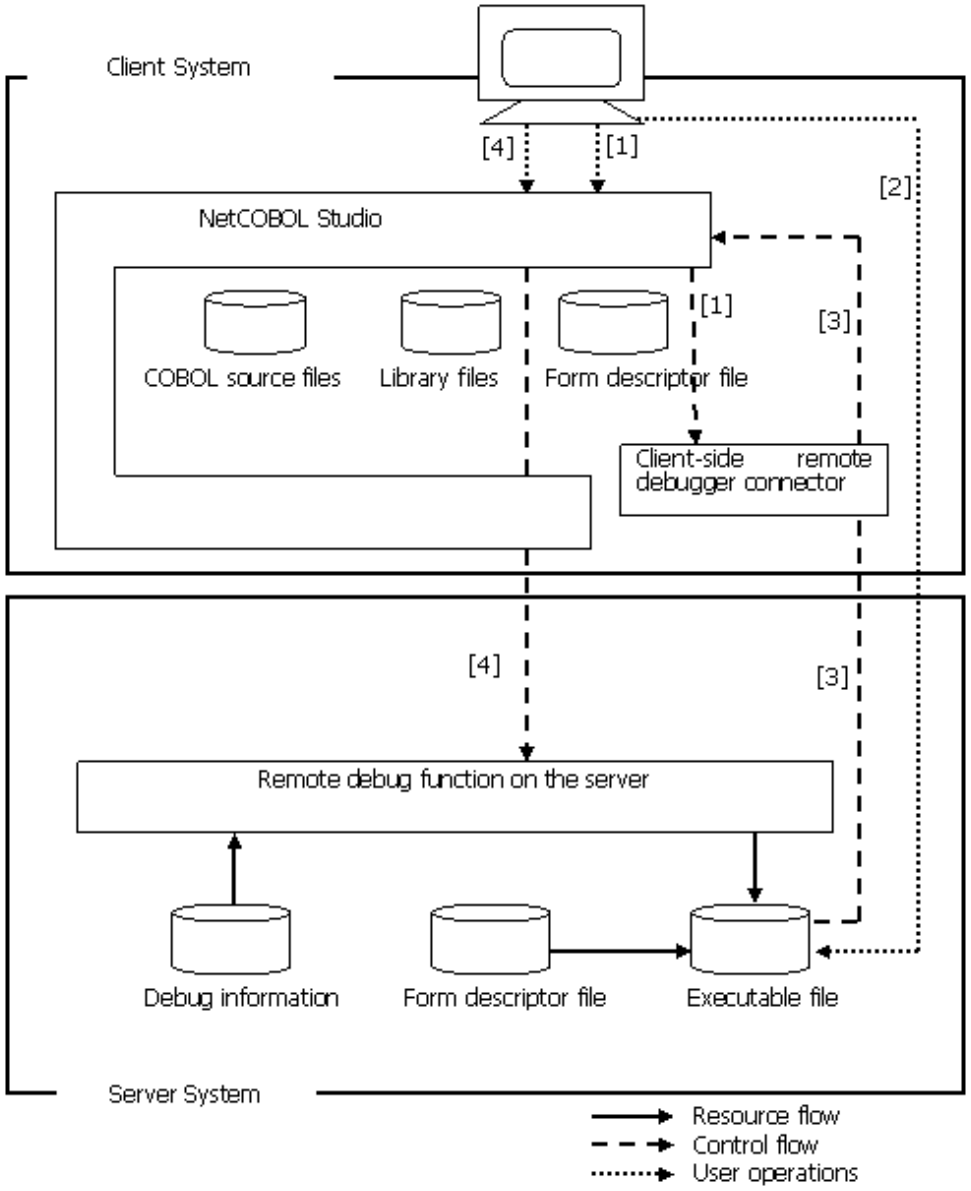
Figure 18.1 Debugging an ordinary program



- [1] For manual operation, start the server-side remote debugger connector. Refer to "18.1.5 Server-side remote debugger connector".
- [2] Use NetCOBOL Studio to start the remote debug function. The remote debug function on the server is started via the server-side remote debugger connector.

- [3] The executable file on the server is debugged by the debug operations at NetCOBOL Studio.

Figure 18.2 Debugging programs that run in a server environment such as Interstage



- [1] Start attach debugging at NetCOBOL Studio. The client-side remote debugger connector is started automatically and NetCOBOL Studio changes to the debug wait state. Refer to "18.1.6 Client-side remote debugger connector".
- [2] Check that the environment variable CBR_ATTACH_TOOL is set at the server, and then execute the executable file. Refer to "18.1.4 CBR_ATTACH_TOOL"
- [3] The executable file can be accessed from NetCOBOL Studio via the client-side remote debugger connector, and the remote debug function is started.
- [4] Use the debug operations at NetCOBOL Studio to debug the executable file on the server.

18.1.4 CBR_ATTACH_TOOL

To use the attach format for remote debugging, the environment variable CBR_ATTACH_TOOL must be set before the COBOL program is executed. CBR_ATTACH_TOOL specification format is as follows:

Specification Format

```
CBR_ATTACH_TOOL=connection-destination/STUDIO [additional-path-list]
```

Connection destination

Specify the port number and the operating computer of the client-side remote debugger connector in the following format:

{ IP_address / Host_name } [: Port_number]

Specify the IP address in either IPv4 or IPv6 format.

Specify a numeric in the 1024 to 65535 range as the port number. If the port number is omitted, a specification of 59999 is used.

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

If a port number is specified in an IPv6 address, enclose the address part in square brackets ([]).

STUDIO

STUDIO is always set as the character string indicating debugging using NetCOBOL Studio.

Additional path list

Specify the debugging information search folder.

The debugging information files are searched in the following sequence and used for debugging:

1. Additional path list specification sequence. (If more than one folder is specified, enter folders separated by a colon ":".)
2. The current folder at the time the COBOL program was first run
3. The storage folder of the COBOL program being started

The current folder at the time the COBOL program being started was first run and the storage folder of the COBOL program being started need not be entered in the additional path list.

Note

If the connection destination is specified in IPv6 format, the system must support IPv6 addresses.

Example

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

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

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

[Specification using 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

[Specification using host name (client-1) and port number (2000)]
        CBR_ATTACH_TOOL=client-1:2000/STUDIO

[Specification with port number omitted (IPv4 address)
        CBR_ATTACH_TOOL=192.168.0.1/STUDIO

[Specification with port number omitted (IPv6 address)]
        CBR_ATTACH_TOOL=fe80::1:23:456:789a/STUDIO
```

18.1.5 Server-side remote debugger connector

In order to perform remote debugging, a program that monitors debug start instructions sent from another computer on the network is required.

A server-side remote debugger connector operates on the server side and monitors debug start instructions sent from the client side.

This is used in the "Method for debugging ordinary programs" described under "18.1.2 Types of remote debugging".

18.1.5.1 Server-side remote debugger connector start method

Start Format

```
svdrds [-p port-number] [connection-restrictions-specification]
```

Port number

Specify a numeric in the 1024 to 65535 range as the port number. If the port number is omitted, a specification of 59998 is used.

Connection restrictions specification

Use the following format to specify settings that restrict the computers that are permitted to connect:

Refer to the notes on using the server-side remote debugger connector in the "NetCOBOL User's Guide" included in the Windows edition of NetCOBOL for details of the entry format for connection restriction specifications and the connection restrictions file.

18.1.5.2 Server-side remote debugger connector termination method

To terminate the server-side remote debugger connector, press the Ctrl key and C key at the same time at the command line that started the remote debugger connector.

18.1.6 Client-side remote debugger connector

In order to perform remote debugging, a program that monitors debug start instructions sent from another computer on the network is required.

A client-side remote debugger connector operates on the client side and monitors debug start instructions sent from the server side.

This is used in the "Method for debugging programs, such as Interstage, that run in a server environment" described under "18.1.2 Types of remote debugging".

18.1.6.1 Client-side remote debugger connector start method

In order to perform the attach type of remote debugging, put NetCOBOL Studio into the state that waits for remote debug start. At this stage, the client-side remote debugger connector starts automatically.

The following icon is displayed in the task tray when the client-side remote debugger connector has started:

Use the [Remote debugger connector] dialog to manage connection destination IP addresses, port numbers, connection restriction specifications, and so on. To open the [Remote debugger connector] dialog, select "Environment settings" from the context menu of the task tray icon.

Refer to the notes concerning the setting method for the [Remote debugger connector] dialog in the "NetCOBOL User's Guide" included in the Windows edition of NetCOBOL for details of the [Remote debugger connector] dialog and the specification method for connection restrictions.

18.1.6.2 Client-side remote debugger connector termination method

To terminate the client-side remote debugger connector, select "End remote debugger connector" from the icon context menu.

Note that the client-side remote debugger connector does not terminate automatically when the remote debug function of NetCOBOL Studio is terminated.

When remote debug is terminated, also terminate the remote debugger connector.

18.1.7 Notes

The following notes apply to the use of the remote debug function of NetCOBOL Studio :

Restriction when a signal is issued

If a signal such as a zero divisor is issued (except for an interrupt signal) during debug processing, subsequent debug tasks (debug program execution) cannot be performed.

Program name lengths

Valid program names consist of a maximum of 4096 alphanumeric characters.

Operation when there is no debugging information file

If there is no debugging information file, the debug program ends without starting. If the debug program forms an endless loop, use the kill command to forcibly terminate the debugger process.

Debugging embedded SQL statements

Breakpoints can be set in an "EXEC SQL" entered in PROCEDURE DIVISION. However, breakpoints cannot be set in an "EXEC SQL" that includes a non-executable statement (WHENEVER or similar). Breakpoints cannot be set directly in an SQL statement.

Remote debug function conditions of use

The server and the client must both support TCP/IP protocol in order for the remote debug function to be used.

18.2 How to Use gdb Command

Although gdb command does not support the COBOL language, this compiler outputs debugging information (source and data position information) for gdb so that the command can debug COBOL programs. This enables debugging a mixture of languages, including C, without changing the debugger.

Although gdb command cannot handle COBOL-unique data types that do not exist in C, this compiler provides user-defined commands to display COBOL-unique data types.

Not all functions of gdb command work normally based on the debugging information output by this compiler. If debugger use methods not described here are used, gdb command may not function properly.

This chapter contains information as listed below. For more information on gdb command functions and their usage, see gdb documents (man gdb, gdb help subcommand output, etc.)

18.2.1 Overview of gdb Command

Use gdb command to detect logical errors of the program while you are running the program. Gdb command directly debugs executable programs. Therefore, you can operate actual business applications using the executable program for debugging and can readily troubleshoot problems found during the operation.

You can use gdb command in any of the following three modes depending on what is to be debugged.

- Debugging an executable program
- Analyzing a core file
- Debugging a process in execution

You need to start gdb command differently for each of these modes. Some functions may not be used depending on what is to be debugged. The basic operation is the same for any of these modes.

18.2.1.1 Debugging an executable program

In this mode, gdb command starts an executable program and debugs it while controlling and displaying the states of the program. This mode is mainly used for testing programs, such as route checking and data verification, during development. See "18.2.4.1 Starting gdb Command to debug an executable program" for information on how to start gdb command in this mode.

18.2.1.2 Analyzing a core file

In this mode, gdb command executes debugging by displaying the state of a core file in the environment in which the core file has been generated. This mode is mainly used for troubleshooting during operation. Analyzing a core file enables you to check the state of a core file at the moment it has been created. See "18.2.4.2 Starting gdb Command to analyze a core file" for information on how to start gdb command in this mode.

18.2.1.3 Debugging a process in execution

In this mode, gdb command debugs a process already in execution. This mode is used when the above two modes are not effective for debugging, such as for debugging a program that is started from another program. See "18.2.4.3 Starting gdb Command to debug a process in execution" for information on how to start gdb command in this mode.

18.2.2 Preparation for Debugging

This section describes to enable effective debugging.

18.2.2.1 Environment settings for gdb Command

The gdb command cannot handle COBOL-unique data types that do not exist in C. Therefore, the command handles the COBOL-unique data types as the character type. This compiler provides user-defined commands to display the COBOL-unique data types. Environment settings are required to use user-defined commands. The following explains how to set an environment for using user-defined commands.

Setting up gdb command initialization file

When issued, gdb command performs initialization by reading the initialization file ".gdbinit" from the user HOME directory.

The package of this compiler includes the "gdbinit.sample" file in the /opt/FJSVcbl64/config directory.

Refer to this file and create initialization file ".gdbinit". If an initialization file already exists, add the contents of "gdbinit.sample" to the initialization file, and then execute gdb command. You can then use user-defined commands.

18.2.2.2 Effects of compile options and source program coding on debugging

This section describes the compile options specified for COBOL program compilation and source program coding that may affect debugging.

OPTIMIZE/NOOPTIMIZE

This is a global optimization option. This option affects the output of object code. Fujitsu recommends specifying NOOPTIMIZE when executing debugging.

If OPTIMIZE is enabled (default), execution results may not be stored in an area, or a statement may be moved or deleted by optimization processing (see "Appendix C Global Optimization"). To debug an object for which OPTIMIZE is enabled, source program information alone is insufficient. Knowledge of assembler is also required for correct debugging. Output target program listing information during compilation to help you during debugging. (See "3.1.7.5 Target Program Listing" for more information.)

NUMBER/NONUMBER

This option relates to the specification of a sequence number area of a source program. This option affects information on the lines output as debugging information. If NUMBER is enabled, gdb command cannot display the correct position of the source program. Fujitsu recommends specifying NONUMBER (default) when executing debugging.

Information

The gdb command cannot use a sequence number to specify line information. Output source program listing information during compilation to ensure that the line numbers output by gdb command match the line numbers in the source program listing. (See "3.1.7.4 Source Program Listing" for more information.)

THREAD(MULTI)/THREAD(SINGLE)

This option relates to the specification for creating a multithread program. A program that has been compiled with THREAD(MULTI) may run in multithread mode. Since a problem may have occurred in a thread other than the one being examined, refer to the thread information to verify the thread that should be examined. For information on displaying thread information or changing the thread to be debugged, see the gdb documents (man gdb, gdb help subcommand output, etc.).

COPY and REPLACE statements

If a COPY or REPLACE statement is specified, some type of processing may be performed during source program compilation, and a mismatch with source program lines may occur. In this case, the results of compilation can be checked by outputting source program listing information in advance during compilation. (See "3.1.7.4 Source Program Listing" for more information.) Fujitsu recommends outputting a source program list in advance when debugging.

18.2.2.3 Resources required for debugging

This section describes the resources required for debugging.

Table 18.2 Resources required for debugging

Resource name	Debug target			Related compile option
	Execution program	Core file	Process in execution	
Execution program file	Yes	Partially (*1)	Yes	
Core file	No	Yes	No	
Resource required for execution	Yes	No	Yes	
Source program file	Partially (*2)			
Source program list	Partially (*3)			SOURCE
Object program list	Partially (*4)			LIST
Data area list	Partially (*5)			MAP

*1 As the core file contains only data, many debugger functions are restricted because an executable program file does not exist.

*2 Required for debugging using a source program.

*3 Required for debugging a program that has been compiled with compile option NUMBER enabled or a program that uses a COPY or REPLACE statement.

*4 Required for debugging a program that has been compiled with compile option OPTIMIZE enabled.

*5 Required for debugging a program that uses data in the based-storage section, data in which an OCCURS clause is specified, data items with the same name, data items without a name, or FILLER item data.

18.2.2.4 Notes on creating programs that can be debugged easily

- Do not create data items with the same name, even if the names differ in the use of uppercase and lowercase letters.
- Use alphanumeric characters (do not include a hyphen "-") to specify program names or data names.

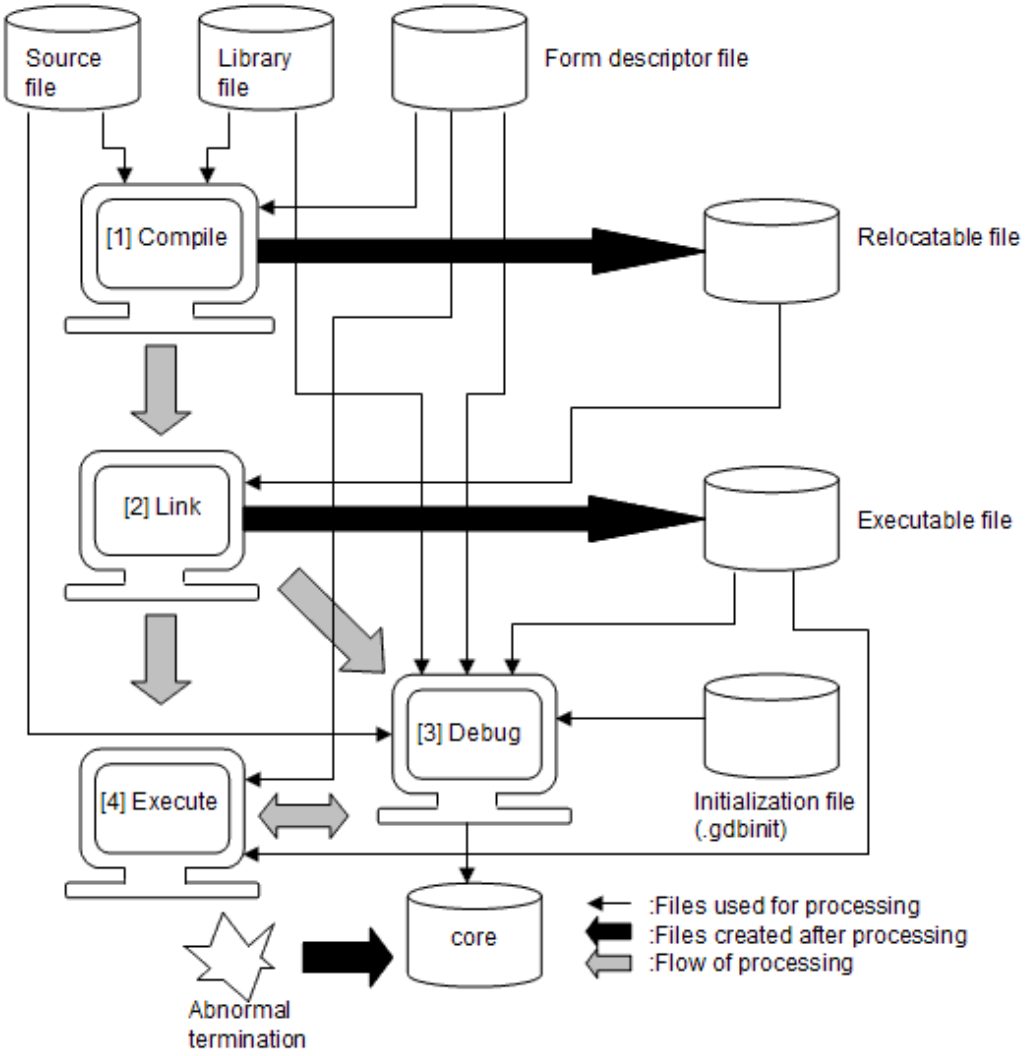
18.2.2.5 Notes on using gdb Command

- Only some functions of gdb command (setting or deleting a breakpoint at a source line number, and referencing data) are supported.

- Setting data values is not supported for the following reasons:
  The "set" subcommand can set a value only for an area that has been allocated as a data area. However, in actual program execution, the value of a data item may be retained in the register. Therefore, even if the value of an area is changed, the original value retained in the register is used, and an expected operation may not be executed. In this case, the value in the register must also be changed.
- For data that can be referenced, see "18.2.5.6 Displaying data (expression)".
- Only data that is used by the program or method currently in execution can be referenced. The data used by other programs and methods cannot be referenced.
- To specify a data name including a hyphen "-", enclose the name in single quotation marks (').
- This compiler unconditionally outputs all debugging information. To exclude debugging information, use the "strip" command. For details of the "strip" command, see the documents for the "strip" command. Debugging is difficult with an executable program file from which debugging information and symbol information have been deleted using the "strip" command. Back up the executable program file before executing the "strip" command.

18.2.3 Procedure for Debugging

This section explains how to debug a COBOL program using gdb command.



- [1] Compile

  When a program is compiled as a debug program, debugging information for gdb is unconditionally output. Collect a source program list, object program list, and data area list as required.

- [2] Link

  The program is linked as a debug program. No specific operation is required.

- [3] Debug

  Debugging is performed using an environment required for execution and resources including an executable program and source program.

- [4] Execute

  The program that has been debugged is actually executed. If the source program is modified, it must be recompiled and re-linked. If the execution ends abnormally and a core file is created, perform debugging in [3] as required.

18.2.4 Starting gdb

You can start debugging in any of three modes depending on what is to be debugged. For details on starting gdb command, see gdb documentation for your operating system.

18.2.4.1 Starting gdb Command to debug an executable program

To debug an executable program, specify the executable program as a gdb argument.

```
$ gdb program
```

program is the file name of the executable program.

18.2.4.2 Starting gdb Command to analyze a core file

To analyze a core file, specify the executable program and core file as gdb arguments.

```
$ gdb program core
```

program is the file name of the executable program. *core* is the core file to be analyzed.

- The core file contains only information on the data that can be modified. The executable program file that was being executed when the core file was generated is required to analyze the core file.
- If the executable program file is re-created, debugging will not be executed normally.

18.2.4.3 Starting gdb Command to debug a process in execution

To debug a process in execution, specify the executable program and process ID as gdb arguments. In this case, a file with a name consisting of the same numeric string as that specified for the process ID must not exist.

```
$ gdb program process-id
```

program is the file name of the executable program. *process-id* is the process ID of the process to be debugged.

- See the description about the ps command for information on how to check the process ID.
- Be careful when specifying the process ID, since specifying a wrong process ID may cause unexpected results.

18.2.5 gdb Command Operation

Use subcommands to operate gdb command. This section explains the subcommands that are often used for debugging.

Function	Subcommand	Explanation
Setting a breakpoint	break [file:]line-number	Sets a breakpoint on the specified source file (file) line.
Listing breakpoints	info breakpoints	Lists the breakpoints that have been set.
Deleting a breakpoint	delete [breakpoint-number]	Deletes the specified breakpoint.
Starting execution	run [argument-list]	Starts running the program. If argument-list is specified, it is passed as an argument to the program.
Restarting execution	continue	Restarts running the program.
	next	Executes program statements in the currently running program one step at a time, excluding statements from other programs.
	step	Executes program statements one step at a time, including the statements from other programs.
Displaying data (expression)	print [/format] expression	Displays the value of an expression.
Displaying a list of data	info locals	Displays a list of effective data items.
Displaying memory data	x [/format] expression	Displays memory data based on the calculation results (address) of an expression.
Displaying a call stack	backtrace	Displays information on the program call stack.
Displaying a source file	list [{- | [file:]line-number}]	Displays a source file (file).
Help	help [name]	Displays help information on gdb. If "name" is specified, help information on the name subcommand is displayed.
Quitting gdb	quit	Quits gdb.

This compiler also provides the user definition commands shown below. Refer to "18.2.2.1 Environment settings for gdb Command" for the environment settings for using the user definition commands.

Table 18.3 User definition commands

Function	Subcommand	Explanation
Displaying data (expression)	binary data-size expression (*1)	Displays COBOL binary items (BINARY, COMP).
	zone data-size expression ext-dec data-size expression (*1)	Displays COBOL external decimal items.
	zonet data-size expression (*1)	Displays COBOL external decimal items (SIGN TRAILING: Default value).
	zonel data-size expression (*1)	Displays COBOL external decimal items (SIGN LEADING).
	zonets data-size expression (*1)	Displays COBOL external decimal items (SIGN TRAILING SEPARATE).
	zonels data-size expression (*1)	Displays COBOL external decimal items (SIGN LEADING SEPARATE).
	pack data-size expression int-dec data-size expression (*1)	Displays COBOL internal decimal items.
	bit data-size expression int-bool data-size expression (*2)	Displays COBOL internal Boolean items.

Function	Subcommand	Explanation
	bool byte-size expression ext-bool byte-size expression (*1)	Displays COBOL external Boolean items.
	national data-size expression utf8 data-size expression (*1)	Displays COBOL national items and national edited items.
	utf16 data-size expression utf16le data-size expression ucs2 data-size expression ucs2le data-size expression (*3)	Displays COBOL national items (UTF-16 little-endian) and national edited items (UTF-16 little-endian).
	utf16be data-size expression ucs2be data-size expression (*3)	Displays COBOL national items (UTF-16 big-endian) and national edited items (UTF-16 big-endian).
	Utf32 data-size expression Utf32le data-size expression (*3)	Displays COBOL national items (UTF-32 little-endian) and national edited items (UTF-32 little-endian).
	Utf32be data-size expression (*3)	Displays COBOL national items (UTF-32 big-endian) and national edited items (UTF-32 big-endian).
	db start-address end-address db start-address length	Dumps the specified memory range.
	CAT address	Displays the contents of the CONTROL ADDRESS TABLE.
Information output when COBOL terminates abnormally	cobcorechk	Outputs the state when terminating abnormally
	cobcorechkex	Outputs detailed COBOL runtime environment information.
	cobcorechkexlist	Outputs COBOL runtime environment information in the form of a simple list.
	cobcorechkexout	Overwrites the "cobcorechkex" results in the "cobcorechkex.txt" file during saving.
	cobcorechkexlistout	Overwrites the "cobcorechkexlist" results in the "cobcorechkexlist.txt" file during saving.

*1: The data-size unit is bytes.

*2: The data-size unit is bits.

*3: The data-size is the number of characters. The UCS2 range of characters can be displayed.

18.2.5.1 Setting a breakpoint

This subcommand specifies the position at which execution is to be stopped. A line number in the source program can be specified.

```
break [file:]line-number
```

file is the file name of a source program. *line-number* is the relative line number in the source program.

- Program execution may not be stopped at the specified line because, for example, an executable statement does not exist at that line. In this case, set another breakpoint at a nearby statement where program execution can be stopped.

- Gdb command does not work normally if a breakpoint is specified on the first line of a COBOL source program. Specify a breakpoint on a line after "PROCEDURE DIVISION."
- For a program that has been compiled with compile option NUMBER specified, a line number in the source program list that has been output during compilation must be specified. If it is not specified, gdb command cannot display the correct position in the source file.

Example

```
(gdb) break DBGSUB.cob:14
Breakpoint 1 at 0x4000000000003541: file DBGSUB.cob, line 14.
```

18.2.5.2 Listing breakpoints

This subcommand displays the current settings of breakpoints.

```
info breakpoints
```

Example

```
(gdb) info breakpoints
Num Type           Disp Enb Address            What
1   breakpoint     keep y   0x4000000000003541 in DBGSUB at DBGSUB.cob:14
```

18.2.5.3 Deleting a breakpoint

This subcommand deletes the specified breakpoint that was set previously. For the breakpoint to be deleted, specify the relevant number displayed by listing breakpoints (see "18.2.5.2 Listing breakpoints").

If no number is specified, the system asks whether to delete all breakpoints. To delete all breakpoints, enter "y"; otherwise, enter "n."

```
delete [breakpoint-number]
```

breakpoint-number is a number displayed in the list of breakpoints.

Example

```
(gdb) delete 1
```

Information

The "disable" command can be used to temporarily disable a breakpoint. To enable the breakpoint again, use the "enable" command.

18.2.5.4 Starting execution

This subcommand starts to execute the program. If an argument-list is specified, it is passed as an argument to the program.

```
run [argument-list]
```

argument-list is an argument to be passed to the program to be executed.

Note

Before executing gdb command, the runtime environment of the COBOL program must be set up. For details, see "4.1 Setting up the Runtime Environment Information".

Example

```
(gdb) run
```

18.2.5.5 Restarting execution

This subcommand restarts execution that was previously stopped.

```
continue
```

This subcommand executes statements one step at a time only in the currently running program, not in other programs.

```
next
```

This subcommand executes statements one step at a time in all programs, including programs other than the one currently running.

```
step
```

Note

The command may fail to restart program execution depending on how program execution has been stopped.

Example

```
(gdb) continue
```

18.2.5.6 Displaying data (expression)

This subcommand displays data and the results of an expression.

```
print [/format] expression
```

/*format* specifies the display format. *expression* can specify an expression including a data name.

Note

- An error results if the specified data cannot currently be referenced.
- See also "18.2.2.5 Notes on using gdb Command" in this chapter.

Example

```
(gdb) print/x DAYS
$2 = {0x0, 0x0, 0x0, 0x0}
```

Information

The table below lists the data items that can be referenced.

Classification	Item	Data referencing
Declaration section	Based-storage section	B (*1)
	File section	A
	Working-storage section	A
	Local-storage section	A
	Constant section	C
	Linkage section	A
Level	Item 01	A
	Item 02 to 49	A
	Item 66	A
	Item 77	A
	Item 78	C
	Item 88	C
Special register	LINAGE-COUNTER	C
	PROGRAM-STATUS	A
	RETURN-CODE	A
	SORT-STATUS	A
	SORT-CORE-SIZE	A
	LINE-COUNTER	C
	PAGE-COUNTER	C
	CONV-STATUS	A
	CONV-SIZE	A

A: Can be referenced

B: Restricted

C: Cannot be referenced

*1 : To reference data in the based-storage section, determine the offset of the data to be referenced from the data map list. Then, add this offset to the value of the pointer data to determine the address to be referenced. Display data from the address thus determined.

Note

- For a data item for which an OCCURS clause is specified, only the first table element can be displayed. To reference a second or subsequent element, determine the offset of the element to be referenced from the data map list and add it to the start address of the data to determine the address to be referenced. Display table element data from the address thus determined.
- If two or more data items have the same name, only information on the data item that is displayed at the top by the info local command is displayed. To reference other data, determine the address of the data to be referenced from the data map list and display the data from the address thus determined.
- Data items without a name and FILLER item data cannot be displayed. To reference these types of data items, determine the address of the data to be referenced from the data map list and display the data from the address thus determined.

COBOL-unique data types that are not supported by gdb are all handled as the character type.

This compiler provides user-defined commands to display the COBOL-unique data types.

Refer to "data attributes symbol" that is output in "Data map listing" in "3.1.7.6 Data Area Listings", and select a display command from the following table.

Data attribute	Data attribute code	Encoding form	Display command (alias)
Fixed-length group item	GROUP-F	-	print (*1) (*2)
Variable-length group item	GROUP-V	-	print (*1) (*2)
Alphabetic character	ALPHA	UTF8	print (*2)
Alphanumeric character	ALPHANUM	UTF8	print (*2)
Alphanumeric edited	AN-EDIT	UTF8	print (*2)
Numeric edited	NUM-EDIT		print (*2)
Index data	INDEX-DATA		print
Zoned decimal	EXT-DEC	-	zone (ext-dec) (*3) (*4)
Zoned decimal (SIGN TRAILING: default)			zonet (*3)
Zoned decimal (SIGN LEADING)			zonel (*3)
Zoned decimal (SIGN TRAILING SEPARATE)			zonets (*3)
Zoned decimal (SIGN LEADING SEPARATE)			zonels (*3)
Packed decimal	INT-DEC	-	pack (int-dec) (*3)
Double-precision internal floating point	FLOAT-L	-	print
Single-precision internal floating point	FLOAT-S	-	print
External floating point	EXT-FLOAT	-	print
Binary number (BINARY, COMP)	BINARY	-	binary (*3)
Binary number (COMP-5, BINARY-XXX (*5))	COMP-5	-	print (*6)
Index name	INDEX-NAME	-	print (*2)
Internal Boolean	INT-BOOLE	-	bit (int-bool) (*3)
External Boolean	EXT-BOOLE	-	bool (ext-bool) (*3)
National (UTF-16 little endian)	NATIONAL	UTF16LE	utf16 (utf16le) (*3)(*7)
National (UTF-16 big endian)		UTF16BE	utf16be (*3)(*7)
National (UTF-32 little endian)		UTF32LE	Utf32 (utf32le) (*3)
National (UTF-32 big endian)		UTF32BE	Utf32be (*3)
National edited (UTF-16 little endian)	NAT-EDIT	UTF16LE	utf16 (utf16le) (*3)(*7)
National edited (UTF-16 big endian)		UTF16BE	utf16be (*3)(*7)
National edited (UTF-32 little endian)		UTF32LE	Utf32 (utf32le) (*3)
National edited (UTF-32 big endian)		UTF32BE	Utf32be (*3)
Pointer item	POINTER	-	print
Object reference data	OBJ-REF	-	Cannot be referenced.

*1 : If data includes a non-character attribute item, display the data in hexadecimal notation (print/x).

*2 : In a Unicode program, 3-byte character data may not be displayed normally. In this case, you may use the user-defined command utf8 to display the data normally.

*3 : User-defined command

*4 : Every type of zoned decimal can be displayed. However, the numeric format may not be checked correctly. If the numeric format is checked, use the command corresponding to the respective sign specification.

*5 : BINARY-CHAR UNSIGNED, BINARY-SHORT [SIGNED], BINARY-LONG [SIGNED], or BINARY-DOUBLE [SIGNED]

*6 : Use hexadecimal notation (print/x) to display data items with a length other than 2, 4, or 8 bytes.

*7 : Display in hexadecimal (print/x) if surrogate pairs are included.

Note

Index data has an offset from the beginning of the table. The index value can be determined by dividing the index data by the size of table elements and then adding 1 to the result.

Example

The following example shows a part of the map list and command input. Select a user-defined command according to the "attribute" in the data map list and specify "length" and "name" as arguments. Specify the command string in the "user-defined-command-name length data-name [+displacement]" format.

Data map list

```
LINE     ADDR    OFFSET REC-OFFSET LVL NAME LENGTH ATTRIBUTE BASE    DIM
   6  heap+000001F0              0 01 TOTAL      9 EXT-DEC  BHG.000000
   9 [heap+00000198]+00000000    0 01 DAYS       4 BINARY   BVA.000001
```

Command

```
(gdb) binary 4 DAYS
0 (0x00000000)
(gdb) zone 9 TOTAL
+0 (0x303030303030303040)
```

18.2.5.7 Displaying a list of data items

This subcommand displays a list of effective data items.

```
info locals
```

Note

The display format cannot be specified. The command displays the results of the print command concerning each type of data.

Example

```
(gdb) info locals
PROGRAM-STATUS = 0
RETURN-CODE = 0
TALLY = 0
ONE-WEEK = "004@000@000R002E005@006E005@"
MON = "004@"
TUE = "000@"
WED = "000R"
THU = "002E"
FRI = "005@"
SAT = "006E"
SUN = "005@"
TMP-DAY = "004@"
```

```
TOTAL = "00000000@"
IDX = 0
DAYS = "\000\000\000"
AVE = "\000"
```

18.2.5.8 Displaying the contents of memory

This subcommand displays the contents of memory. The result of the expression is handled as an address.

```
x [/format] expression
```

/format specifies the display format. *expression* can specify an expression including a data name.

For information on the data representation in memory, see "General Rules" in Section 5.4.16, "USAGE Clause" in the "NetCOBOL Language Reference".

Example

```
(gdb) x/9bx TOTAL
0x6000000000014e40:  0x30  0x30  0x30  0x30  0x30  0x30  0x30  0x30
0x6000000000014e48:  0x40
```

18.2.5.9 Displaying the contents of the call stack

This subcommand displays the calling path from the current breakpoint to the program that was first called.

```
backtrace
```

Note

If there is an internal subprogram in the calling path, the calling path is not correctly displayed. If the calling path needs to be correctly displayed for debugging, temporarily change the internal subprogram to an external program before starting debugging.

Example

```
(gdb) backtrace
#0  0x4000000000003d10 in DBGSUB () at DBGSUB.cob:19
#1  0x4000000000001e80 in DBGMAIN () at DBGMAIN.cob:18
#2  0x4000000000000c70 in DBGMAIN () at DBGMAIN.cob:1
```

18.2.5.10 Change of stack frame

This subcommand changes the stack frame.

```
frame frame-number
```

This specification changes to the stack frame of the specified frame-number.

Refer to "18.2.5.9 Displaying the contents of the call stack " for the frame number.

```
up [frame-count]
```

This specification changes to the stack frame of the frame number in which frame-count is added to the frame number for a present stack frame. It is considered that one is specified when frame-count is omitted.

```
down [frame-count]
```

This specification changes to the stack frame of the frame number in which frame-count is pulled from the frame number for a present stack frame. It is considered that one is specified when frame-count is omitted.

Example

- When changing from the stack frame (#0) of the current breakpoint to the stack frame (#1) of the calling program.

```
(gdb) up
#1 0x0000000000400db8 in DBGMAIN () at DBGMAIN.cob:18
18 000018 CALL "DBGSUB" USING DAYS RETURNING AVE
```

- When changing to an arbitrary stack frame (#1).

```
(gdb) backtrace
#0 0x0000000000401813 in DBGSUB () at DBGSUB.cob:19
#1 0x0000000000400db8 in DBGMAIN () at DBGMAIN.cob:18
#2 0x0000000000400954 in main () at DBGMAIN.cob:1
(gdb) frame 1
#1 0x0000000000400db8 in DBGMAIN () at DBGMAIN.cob:18
18 000018 CALL "DBGSUB" USING DAYS RETURNING AVE
```

- When changing from the stack frame (#1) of the called program to the stack frame (#0) of the current breakpoint.

```
(gdb) down
#0 0x0000000000401813 in DBGSUB () at DBGSUB.cob:19
19 000019 COMPUTE AVE = TOTAL/ DAYS
```

18.2.5.11 Displaying a source file

This subcommand displays the source program from near a breakpoint where program execution is currently stopped or from the specified line number.

```
list [{- | [file:]line-number}]
```

"-" indicates that the lines before the breakpoint where program execution is currently stopped be displayed. *file* is the file name of the source program. *line-number* is a relative line number in the source program.

Example

```
(gdb) list
14    000014     PERFORM DAYS TIMES
15    000015         ADD  1  TO  IDX
16    000016         ADD  TMP-DAY (IDX)  TO  TOTAL
17    000017     END-PERFORM
18    000018*
19    000019     COMPUTE  AVE = TOTAL / DAYS
20    000020*
21    000021     EXIT PROGRAM
22    000022 END PROGRAM DBGSUB.
```

18.2.5.12 Help

This subcommand displays information such as the subcommands that can be used.

```
help [name]
```

name specifies a subcommand or user-defined command.

Example

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

18.2.5.13 Quitting gdb Command

This subcommand quits gdb command.

```
quit
```

Example

```
(gdb) quit
$
```

If you do not want to quit the process to be debugged, use the "detach" subcommand.

18.2.6 Examples of Debugging

This section provides examples of debugging the following program.

EXTDATA.cbl

```
000001 01  ONE-WEEK USAGE DISPLAY IS EXTERNAL.
000002  02  MON  PIC S9(3)V9.
000003  02  TUE  PIC S9(3)V9.
000004  02  WED  PIC S9(3)V9.
000005  02  THU  PIC S9(3)V9.
000006  02  FRI  PIC S9(3)V9.
000007  02  SAT  PIC S9(3)V9.
000008  02  SUN  PIC S9(3)V9.
000009 01 REDEFINES ONE-WEEK.
000010  02  TMP-DAY  PIC S9(3)V9 USAGE DISPLAY OCCURS 7 TIMES.
```

DBGMAIN.cob

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. DBGMAIN.
000003 DATA DIVISION.
000004 WORKING-STORAGE SECTION.
000005 01  DAYS     PIC 9(9)    USAGE BINARY.
000006 COPY EXTDATA.
```

```
000007 01  AVE      PIC S9(3)V9 USAGE BINARY.
000008 01  AVE-TEMP PIC ++9.9   USAGE DISPLAY.
000009 PROCEDURE DIVISION.
000010     MOVE  4.0  TO  MON
000011     MOVE  0.0  TO  TUE
000012     MOVE -0.2  TO  WED
000013     MOVE  2.5  TO  THU
000014     MOVE  5.0  TO  FRI
000015     MOVE  6.5  TO  SAT
000016     MOVE  5.0  TO  SUN
000017*
000018     CALL "DBGSUB" USING DAYS RETURNING AVE
000019*
000020     MOVE AVE TO AVE-TEMP
000021     DISPLAY "AVERAGE TEMPERATURE OF THIS WEEK IS" AVE-TEMP "DEGREES."
000022     EXIT PROGRAM
000023 END PROGRAM DBGMAIN.
```

DBGSUB.cob

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. DBGSUB.
000003 DATA DIVISION.
000004 WORKING-STORAGE SECTION.
000005 COPY EXTDATA.
000006 01  TOTAL PIC S9(8)V9 USAGE DISPLAY.
000007 77  IDX   PIC S9(9)   USAGE COMP-5.
000008 LINKAGE SECTION.
000009 01  DAYS  PIC 9(9)    USAGE BINARY.
000010 01  AVE   PIC S9(3)V9 USAGE BINARY.
000011 PROCEDURE DIVISION USING DAYS RETURNING AVE.
000012     MOVE  0  TO  TOTAL
000013     MOVE  0  TO  IDX
000014     PERFORM DAYS TIMES
000015         ADD  1  TO  IDX
000016         ADD  TMP-DAY(IDX)  TO  TOTAL
000017     END-PERFORM
000018*
000019     COMPUTE  AVE = TOTAL / DAYS
000020*
000021     EXIT PROGRAM
000022 END PROGRAM DBGSUB.
```

Although this example explains the method of debugging an executable program, the operation to be performed after gdb start is the same as that for other methods.

18.2.6.1 Debugging using the source program

This section provides an example of debugging the above program using the source program.

Compiling the target program

To execute debugging using the source program, compile the target program with compile option NOOPTIMIZE specified.

```
$ cobol -c -WC,"NOOPTIMIZE,SOURCE,COPY,MAP" -P- DBGSUB.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ cobol -M -WC,"NOOPTIMIZE,SOURCE,COPY,MAP" -P- DBGMAIN.cob DBGSUB.o
HIGHEST SEVERITY CODE = I
```

The executable program file "a.out" is created.

Starting gdb

Atart debugger gdb.

```
$ gdb a.out
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
```

Setting a breakpoint

Set a breakpoint at the point to be checked.

```
(gdb) break DBGMAIN.cob:10
Breakpoint 1 at 0x4000000000001be1: file DBGMAIN.cob, line 10.
```

Executing

Run the debug program.

```
(gdb) run
Starting program: /home/debugtest/a.out
[Thread debugging using libthread_db enabled]
[New Thread 2305843009220073984 (LWP 31927)]
[Switching to Thread 2305843009220073984 (LWP 31927)]

Breakpoint 1, DBGMAIN () at DBGMAIN.cob:10
10    000010     MOVE  4.0  TO  MON
```

Checking data

No values are entered because there are no initial values and statements for setting values have not been executed.

```
(gdb) info locals
PROGRAM-STATUS = 0
RETURN-CODE = 0
TALLY = 0
ONE-WEEK = '\0' <repeats 27 times>
MON = "\000\000\000"
TUE = "\000\000\000"
WED = "\000\000\000"
THU = "\000\000\000"
FRI = "\000\000\000"
SAT = "\000\000\000"
SUN = "\000\000\000"
TMP-DAY = "\000\000\000"
DAYS = "\000\000\000"
AVE = "\000"
AVE-TEMP = "\000\000\000\000"
```

Executing one statement

Execute the statement that sets a value for the first data item.

```
(gdb) next
11    000011     MOVE  0.0  TO  TUE
```

Checking the data again

A value is now set because line 10 was executed.

```
(gdb) zone 4 MON
+40 (0x30303440)
```

Setting a breakpoint in the CALL statement and executing it

```
(gdb) break 18
Breakpoint 2 at 0x4000000000001e91: file DBGMAIN.cob, line 18.
(gdb) continue
Continuing.

Breakpoint 2, DBGMAIN () at DBGMAIN.cob:18
18    000018     CALL "DBGSUB" USING DAYS RETURNING AVE
```

Executing the CALL statement

Use the step command to debug the program called by the CALL statement.

```
(gdb) step
DBGSUB () at DBGSUB.cob:2
2       000002 PROGRAM-ID. DBGSUB.
```

Run the program up to PROCEDURE DIVISION

```
(gdb) next
11      000011 PROCEDURE DIVISION USING DAYS RETURNING AVE.
```

Run the program up to the PERFORM statement on line 14

```
(gdb) break 14
Breakpoint 3 at 0x4000000000003651: file DBGSUB.cob, line 14.
(gdb) continue
Continuing.

Breakpoint 3, DBGSUB () at DBGSUB.cob:14
14    000014     PERFORM DAYS TIMES
```

Check the operation of the PERFORM statement

Use the next command to execute a single statement.

```
(gdb) next
19    000019         COMPUTE  AVE = TOTAL / DAYS
```

Line 15 line is not executed and the next execution statement of END-PERFORM in Line 19 line is displayed.

Check the condition of the PERFORM statement

```
(gdb) binary 4 DAYS
0 (0x00000000)
```

The argument "DAYS" is called with the value "0" . This shows that the PERFORM statement has not been executed correctly.

Correct the calling source program and execute debugging again to check for normal operation.

18.2.6.2 Debugging using assembler

If OPTIMIZE is enabled (default), execution results may not be stored in an area, or a statement may be moved or deleted by optimization processing (see "Global Optimization"). In addition, machine instructions may be sorted to execute two or more executable statements in parallel. If stepwise execution is performed, the steps are not executed in the order in which they are coded in the source program. In this event, route check debugging is difficult and, therefore, debugging is performed by checking whether the contents of data at the occurrence of an abnormal end are correct. Knowledge of assembler is required to check data, because data may be stored in registers.

Using assembler, output an object program list in advance during compilation so that you can perform debugging using machine instructions.

For information on the format of the object program list, see "3.1.7.5 Target Program Listing".

Compiling the target program

To debug a program for which compile option OPTIMIZE is enabled, output an object program list by adding the compile operation LIST.

```
$ cobol -c -WC,"OPTIMIZE,SOURCE,COPY,MAP,LIST" -P- DBGSUB.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ cobol -M -WC,"OPTIMIZE,SOURCE,COPY,MAP,LIST" -P- DBGMAIN.cob DBGSUB.o
HIGHEST SEVERITY CODE = I
```

Starting gdb

```
$ gdb a.out
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
```

Executing

```
(gdb) run
Starting program: /home/debugtest/a.out
[Thread debugging using libthread_db enabled]
[New Thread 2305843009220073984 (LWP 25274)]

Program received signal SIGFPE, Arithmetic exception.
[Switching to Thread 2305843009220073984 (LWP 25274)]
0x4000000000003d11 in DBGSUB () at DBGSUB.cob:19
19    000019     COMPUTE  AVE = TOTAL / DAYS
```

An error occurred on line 19 of the source program "DBGSUB.cob".

Checking the instruction that resulted in an error

```
(gdb) x/5i $rip
0x4017fa <DBGSUB+1794>: idiv   r12
0x4017fd <DBGSUB+1797>: mov    r11,rax
0x401800 <DBGSUB+1800>: mov    rcx,r11
0x401803 <DBGSUB+1803>: rol    cx,0x8
0x401807 <DBGSUB+1807>: mov    WORD PTR [rsp+0x1d0],cx
```

A division error (#DE) hardware exception is issued because the divisor is "0". The OS converts this hardware exception to a signal and issues "SIGFPE".

Refer to the "Instruction set reference" published by Intel/AMD for details of machine language instructions.

Confirming an error location from the object program list

Search the <DBGSUB+1794> information, obtained when the machine instructions were checked, for the position where the offset from external label DBGSUB is 1794.

Calculate the position within the intended program list

To calculate the position within the intended program list, perform the following: add the offset of the next instruction of the "DBGSUB" label displayed in the intended program list to the offset from the label output by gdb.

```
(gdb) print/x 0x38+1794
$2 = 0x73a
```

Program list (DBGSUB.lst)

```
                          DBGSUB:    ** External label of DBGSUB **
000000000038  55            push     rbp      <- Offset position of label within .text
   :
--- 19 ---            COMPUTE
0000000006C9  4C8BBB98010000   mov    r15,qword ptr [rbx+0x00000198]  BVA.1
0000000006D0  458B27           mov    r12d,dword ptr [r15]            DAYS
0000000006D3  410FCC           bswap  r12d
0000000006D6  4D63E4           movsxd r12,r12d
0000000006D9  0FB6BBF8010000   movzx  edi,byte ptr [rbx+0x000001F8]   TOTAL+8
0000000006E0  897C2430         mov    dword ptr [rsp+0x30],edi
0000000006E4  488DBC2494000000 lea    rdi,[rsp+0x00000094]            TRLP+0
0000000006EC  48897C2438       mov    qword ptr [rsp+0x38],rdi
0000000006F1  48C7C709000000   mov    rdi,0x00000009
0000000006F8  488DB3F0010000   lea    rsi,[rbx+0x000001F0]            TOTAL
0000000006FF  4C8B93C0010000   mov    r10,qword ptr [rbx+0x000001C0]
000000000706  41FF9200010000   call   qword ptr [r10+0x00000100]
00000000070D  8B7C2430         mov    edi,dword ptr [rsp+0x30]
000000000711  4889C2           mov    rdx,rax
000000000714  48F7DA           neg    rdx
000000000717  4881E7F0000000   and    rdi,0x000000F0
00000000071E  4080FF50         cmp    dil,0x50
000000000722  480F44C2         cmovz  rax,rdx
000000000726  488B7C2438       mov    rdi,qword ptr [rsp+0x38]
00000000072B  8907             mov    dword ptr [rdi],eax
00000000072D  4C63942494000000 movsxd r10,dword ptr [rsp+0x00000094]  TRLP+0
000000000735  4C89D0           mov    rax,r10
000000000738  4899             cqo
00000000073A  49F7FC           idiv   r12                               <- HERE!
00000000073D  4989C3           mov    r11,rax
000000000740  4C89D9           mov    rcx,r11
000000000743  66C1C108         rol    cx,0x08
000000000747  66898C24D0010000 mov    word ptr [rsp+0x000001D0],cx    AVE
```

Software exception "1: integer divided by zero" is generated at "000019 COMPUTE AVE = TOTAL / DAYS" on line 19. The cause of this exception is probably that "DAYS" is 0. It is hard to check it with the OPTIMIZE option enabled. Execute debugging in accordance with "18.2.6.1 Debugging using the source program".

Chapter 19 COBOL File Utility

The COBOL File Utility allows you to use utility commands to execute file processing that can be used with 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, and sort).

Other operations for COBOL files include converting file organization, and reorganizing, recovering, and displaying attributes of indexed files. These utilities can be easily commands.

19.1 Overview

The COBOL File Utility allows you to use utility commands to execute file processing that can be used with COBOL file systems (simply called "COBOL files" hereafter), without using COBOL applications.

- With the COBOL File Utility, you can create COBOL files based on data generated with text editors.
- You can manipulate COBOL files (converting file organization, reorganizing, recovering, and displaying attributes of indexed files).
- You can manipulate COBOL records (such as display, edit, and sort).

The file utility provides a command mode. Command mode enables you to operate a COBOL file with the commands corresponding to each function.

19.2 COBOL File Utility Functions

This section explains the COBOL File Utility functions.

19.2.1 Function Overview

Function	Functional outline
Convert	Converts from a text file to a variable-length record sequential file or vice versa.
Load	Creates or extends a fixed or variable-length record sequential, relative, or indexed files from a variable-length record sequential file.
Unload	Creates a variable-length record sequential file from a fixed or variable-length record sequential, relative, or indexed file.
Browse	Browses the content of records in COBOL files.
Extend	Extends a file.
Sort	Sorts records by an arbitrary key and places the results in a variable-length record sequential file.
Attribute	Displays the attribute information of an indexed file.
Recover	Recovers corrupted indexed files.
Reorganize	Reduce file size by deleting empty blocks in indexed files.

19.2.2 How to Manipulate a File

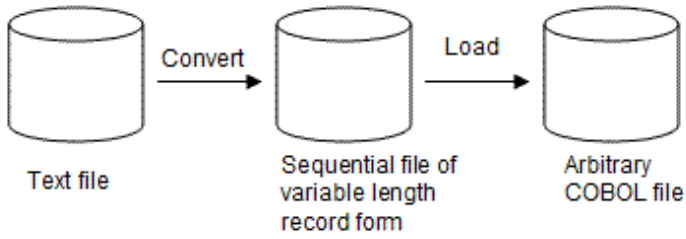
This section explains the basic file operation using the COBOL File Utility functions.

Creating a File

There are two ways to create a file: using a text file or using a COBOL file.

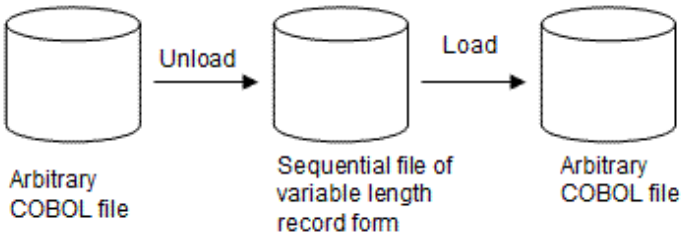
- Using a Text File

  Combining the Convert function with the Load function creates a fixed or variable-length record sequential, relative, or indexed file from a text file created with a text editor.



- Using a COBOL File

  Combining the Unload function with the Load function creates a fixed or variable-length record sequential, relative, or indexed file from an existing fixed or variable-length record sequential, relative, or indexed file.

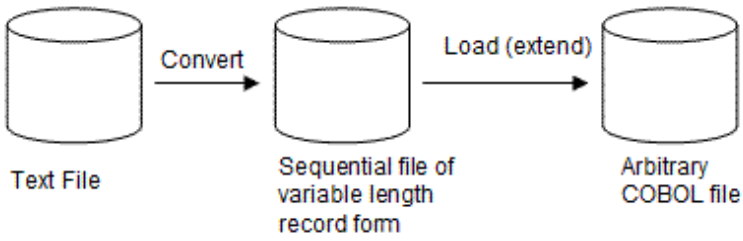


Extending a File

There are some ways to extend (add) a file: using a text file, or using a COBOL file.

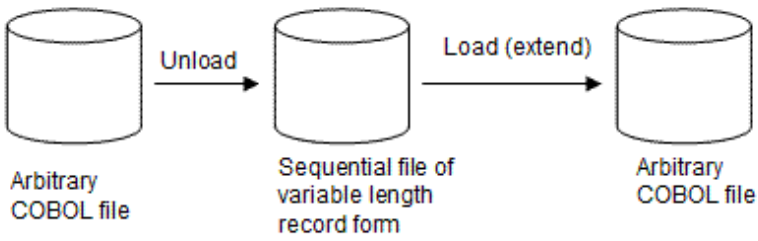
- Using a Text File

  Combining the Convert function with the Load function extends an existing fixed or variable-length record sequential, relative, or indexed file with the content of a text file created with a text editor.



- Using a COBOL File

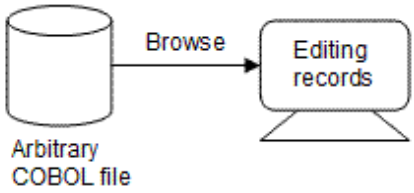
  Combining the Unload function with the Load function extends a fixed or variable-length record sequential, relative, or indexed file with the content of an existing fixed or variable-length record sequential, relative, or indexed file.



Browsing a Record

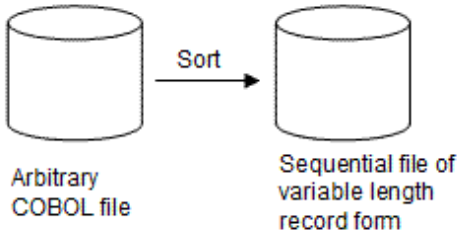
Browse the contents of records in COBOL files with the Browse function.

In command format specifying the display start point, display end point or the number of records to display displays the records within an optional scope.



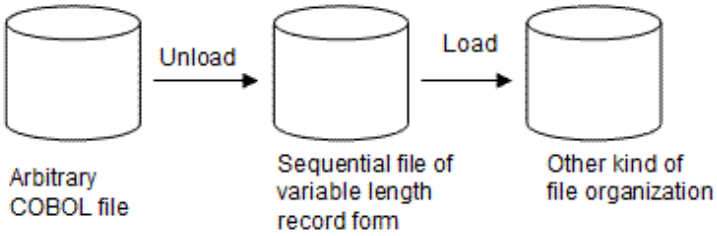
Sorting a Record

Using the Sort function sorts the records in a file by a data item in a record and outputs the sorting result to a variable-length record sequential file.



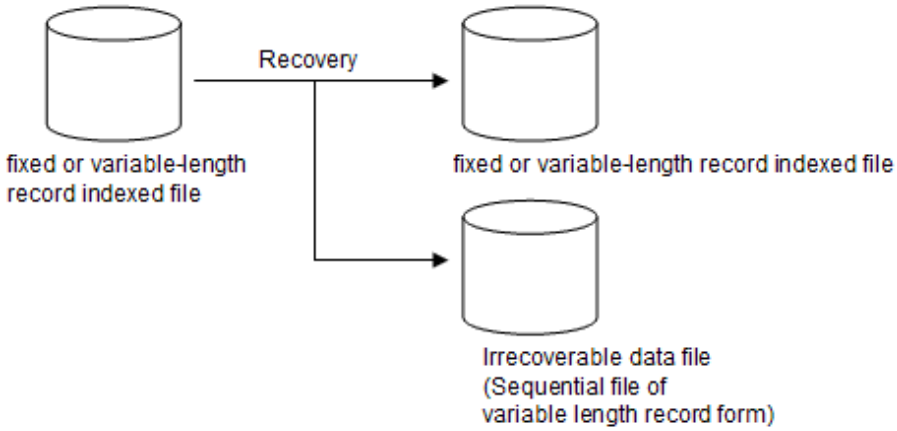
Converting File Organization

Combining the Load function with the Unload function converts a fixed or variable-length record sequential, relative, or indexed file to another file organization.



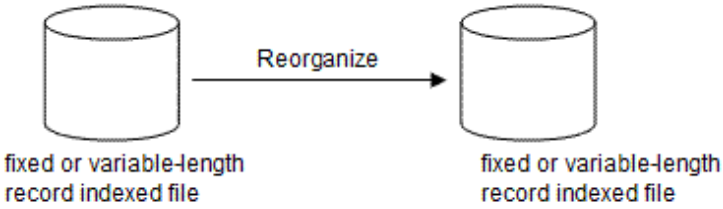
Recovering an Indexed File

A COBOL program may terminate abnormally during opening of an indexed file, resulting in incorrect closing of the file. This may destroy the relationship between records and keys in the indexed file. In this case you can no longer access the indexed file. Using the Recover function recovers this kind of situation. However, if some data is corrupt and some records cannot be recovered, those records are output as an unrecoverable data file in the format of variable-length record sequential file.



Reorganizing an indexed file

Using the Reorganize function deletes as many blank blocks in an indexed file as possible to reduce the size of the file.

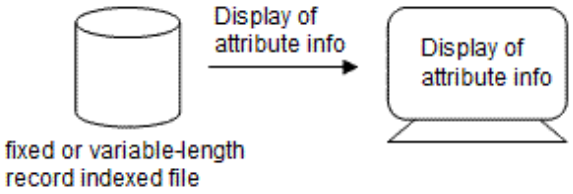


Note

Erasing empty blocks may deteriorate file access performance.

Displaying the Attribute of an Indexed File

Using the Attribute function displays the attribute information (record length, record format, key information, etc.) of an indexed file.



Note

Only attribute information of indexed files is displayed.

Notes on File Manipulation

- You can create a backup file when you use the Edit and Extend functions. The content of the original file can be saved in the backup file and restored if the file becomes corrupt because an error occurred during execution of the COBOL File Utility. It is recommended to create a backup file when you edit or extend a file. A backup file is generated with a name with extension "bak" in the directory where the file is stored.

- When a file is edited or displayed, the file is accessed according to the COBOL syntax. Thus, you cannot insert/delete a record to or from a sequential file. You can access a record for a relative/indexed file sequentially or randomly. However, only sequential access is allowed in a sequential file.
- You cannot update the same record for a sequential or indexed file continuously. If you want to update the same record again, display the record you want to update again.

19.3 How to Use the Command Mode

This section explains how to use the COBOL File Utility in command mode.

19.3.1 Convert

Create a variable-length record sequential file based on a text file or vice versa.

Information

- "Text file" is a line sequential file of COBOL and consists of character and hexadecimal data. One record is data of one line separated by a line feed character.
- "Character data" is the data (character data) that represents characters.
- A 3-byte character "abc" is shown as "abc".
- "Hexadecimal data" is the data that represents a binary data in hexadecimal.
- A 2-byte binary value "5" is shown as "0005".
- A 4-byte binary value "3456" is shown as "00000d80".
- A continuous data of 3-byte character "abc", 2-byte binary "5", and 3-byte character "999" is shown as "abc0005999".
- When environment variable LANG is Unicode, the character code of the text file is treated as UTF-8.

Specification Format

```
cobfconv -ooutput-file-name -cmode,format  input-file-name
```

Parameter

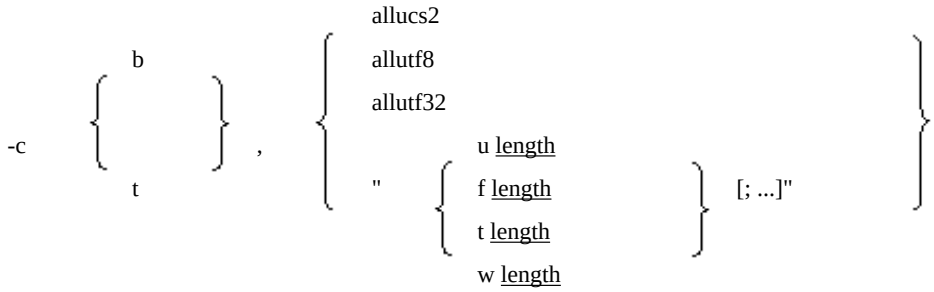
output-file-name

Specify the path name of the text file or the record sequential file to which the conversion result is output.

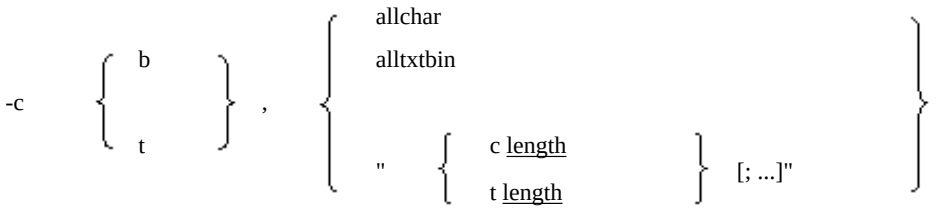
mode, format

Specify the conversion mode and record composition of record sequential file (data format) as shown below:

- The code system is Unicode:



- The code system is Shift-JIS:



Specify the conversion mode using one of the following characters:

- b : Convert a text file to a record sequential file. The data description form of the record is specified.
- t : Convert a record sequential file to a text file. The data description form of the record is specified.
- allchar : All data in the record is treated as character format.
- alltxtbin : All data in the record is treated as hexadecimal format.
- "{c length|t length}[;...]" : Specify the length of the data format next to the keyword character "c" (character format) and "t" (hexadecimal format) when character and hexadecimal data are in the same record. For example, c8;t4 shows 8-byte character with 4-byte hexadecimal.
- allucs2 : All data in the record is treated as UCS-2 format.
- allutf8 : All data in the record is treated as UTF-8 format.
- allutf32 : All data in the record is treated as UTF-32 format.
- "{u length|f length |t length}[;...]" : Specify the length of the data format (the number of characters in case of UCS-2 format) next to the keyword character "u" (UCS-2 format), "f" (UTF-8 format), "w" (UTF-32 format) and "t" (hexadecimal format) when several data formats are in the same a record.

Note

You should note the following when you specify the length:

- Specify the length before the conversion to the hexadecimal format in converting from a record sequential file to a text file. For example, t4 is an 8-byte data in hexadecimal after the conversion.
- Specify the length after the conversion to the character format in converting from a text file to a record sequential file. For example, t4 is an 8-byte data in hexadecimal before the conversion.
- If one record contains a combination of multiple data formats, up to 256 data formats can be specified.

input-file-name

Specify the path name of the text file or record sequential file in the conversion origin.

Example

- Convert a record sequential file to a text file (all Unicode and UTF-8 format):

```
$ cobfconv -ooutfile -ct,allutf8 infile
```

19.3.2 Load

Create a fixed or variable-length record sequential, relative, or indexed file based on a variable-length record sequential file. This command also adds record data in a variable-length record sequential file to an existing 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 -ooutput-file-name [-e] -d file-attribute  input-file-name
```

Parameter

output-file-name

Specifies the path to the file to be created or extended.

-e

Specify this parameter for file extension. You cannot specify the record format, record length, and key information of the "-d" parameter when you extend an indexed file.

file-attribute

Specify the attribute of the file to be created or extended, using the following format:

-d { S / R / I } , { f / v } , record-length, "(offset, length[,N] [,N32] [/offset, length[,N] [,N32]] ...)[,D][; ...]"

Specify the file organization using one of the following characters:

- S : Sequential file
- R : Relative file
- I : Indexed file

Specify the record format using one of the following characters:

- f : Fixed-length
- v : Variable-length
- record-length : Specify the maximum record length when using variable-length records.

To create an indexed file, specify record key information as follows:

- offset : Specify the location-inter-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 that the data item used as a record key is in UTF-16 format.
- N32: Specify that the data item used as a record key is in UTF-32 format.
- / : When using two or more discontinuous data items as one record key, specify individual data items by separating their offset and lengths with a slash (/).
- 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 a semi-colon (;).

input-file-name

Specify the path name of the record sequential file in which the records to create or extend are stored. The file to specify should be a variable-length record sequential file.

Example

To extend an indexed file:

```
$  cobfload -oixdfile -dI,v,80,"(0,5/10,5),D;(5,5)" infile
```

To extend a relative file:

```
$  cobfload -orelfile -e -dR,f,80 infile
```

Note

- In specifying to extend an indexed file you can output the record with the key value less than the maximum key value.
- 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.
- If an error occurs in specifying to extend a file, the content of the output file is what is previous to the file extension.

19.3.3 Unload

Create a variable-length record sequential file based on a fixed or variable sequential, relative, or indexed file.

Specification Format

```
cobfulod -ooutput-file-name -ifile-attribute  input-file-name
```

Parameter

output-file-name

Specify the path name of the variable-length record sequential file to be created.

file-attribute

Specify the attribute of the file to unload, using the following format:

-i { S / R / I } , { f / v } , record-length

Specify the file organization using one of the following characters:

- S : Sequential file
- R : Relative file
- I : Indexed file

Specify the record format using one of the following characters. However, when an indexed file is specified for the file organization, the record format cannot be specified as the attribute information in the file is used.

- f : Fixed-length
- v : Variable-length
- record-length : Specify the maximum record length when using variable-length records. However, if you specify an indexed file as file organization, specification of the record length is not available as the attribute information in the file is used.

input-file-name

Specify the path name of the variable-length record sequential file to unload. The file to specify should be a fixed or variable-length record sequential, relative, or indexed file.

Example

To create a record sequential file from a relative file:

```
$  cobfulod -ooutfile -iR,f,80 relfile
```

Note

- Processing may not take place correctly if the file organization of the input file is different from the actual file.
- Processing does not take place correctly if the record length of the input file is different from the actual file.

19.3.4 Browse

Displays the content of a file in both character format and hexadecimal format in units of records. Data other than alphanumeric characters (0x20...0x7e) is replaced by periods before displayed. 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 -ifile-attribute  [-ps start-location] [-pe end-location] [-po display-order] [-pk search-key-number] input-file-name
```

Parameter

file-attribute

Specify the attribute of the file to be displayed, using the following format:

-i { S | L | R | I } , { f | v } , record-length

Specify the file organization using one of the following characters:

- S : Record sequential file
- L : Line sequential file
- R : Relative file
- I : Indexed file

Specify the record format using one of the following characters. However, when an indexed file is specified for the file organization, the record format cannot be specified as the attribute information in the file is used.

- f : Fixed-length
- v : Variable-length
- record-length : Specify the maximum record length when using variable-length records. However, if you specify an indexed file as file organization, specification of the record length is not available as the attribute information in the file is used.

start-location

Specify the location of the record to start display, using the following format:

-ps { store-order | r relative-record-number | ic record-key-value | it record-key-value }

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". 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:

```
-pe { store-order
    | r relative-record-number
    | 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:

```
-po { A
   | 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, 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. A record key number is a numeric character where the record key number 0 is assigned to the main record key, 1 is assigned to the first sub record key, and 2 and subsequent serial numbers are assigned to the second and subsequent sub record keys. If the search key number is omitted, the main record key is used.

input-file-name

Specify the path name of a file to be displayed.

Example

To display the content of sequential files:

```
$  cobfbrws -iS,v,80 -ps5 -pet10 seqfile
```

To display the content of relative files:

```
$  cobfbrws -iR,f,50 -psr20 -per10 -poD relfile
```

To display the content of indexed files:

```
$  cobfbrws -iI -psit0001 -peit0010 -poA -pk1 ixdfile
```

Note

- Processing may not take place correctly if the file organization of the input file is different from the actual file.

- Processing does not take place correctly if the record length of the input file is different from the actual file.

19.3.5 Sort

Sorts the records in a file in ascending or descending order using a specified data item in the records as a key and the sorted records are output to a variable-length record sequential file.

You can specify a directory in which the sort work file is created with the environment variable BSORT_TMPDIR. When you specify this directory, specify the path name of the directory. If the environment variable BSORT_TMPDIR is not specified, refer to "10.2.4 Program Execution" for the sort work file.

Specification Format

```
cobfsort -ooutput-file-name -ssort-condition -ifile-attribute  input-file-name
```

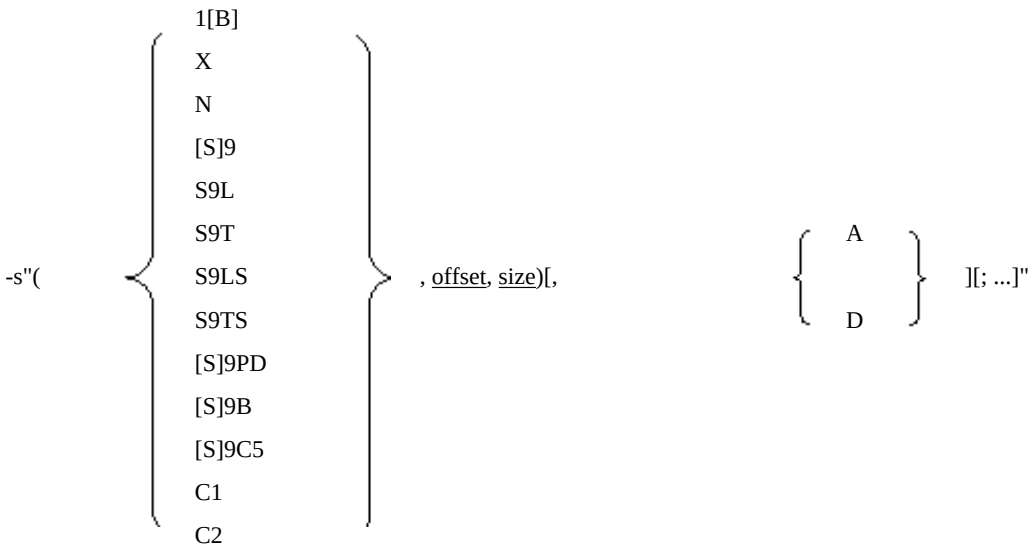
Parameter

output-file-name

Specify the path name of the sequential file to which the sorted records are to be output.

sort-condition

Specify the attribute of the data item to be used as a key and sort order, using the following format:



Specify one of the following values for the key item attribute.

A keyword character of the item attribute to specify is represented as the data attribute of COBOL.

- 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
- 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.

  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. Specify 4 bytes for "C1" and 8 bytes for "C2".

Specify whether records should be sorted in ascending order (keyword character "A") or descending order (keyword character "D") according to the attribute of the key item.

- A : Ascending order
- D : Descending order

Default records are sorted in ascending order.

Information

When "1B" is specified for the data attribute, the sort key value is a logical conjunction of one byte from the position in a specified record and the value specified as a mask value. For example, if "1" is set as offset and "227" (decimal notation) as a mask value, the sort key value is "a1" (hexadecimal) if the content of the record is "05ad" (hexadecimal). This is a logical conjunction of the relative one-byte in the record "ad" (hexadecimal) and the mask value "e3" (hexadecimal notation of "227").

Note

Up to 64 sorting conditions can be specified.

file-attribute

Specify the attribute of the file to be sorted, using the following format:

-i { S / L / R / I } , { f / v } , record-length

Specify the file organization using one of the following characters:

- S : Sequential file
- L : Line sequential file
- R : Relative file
- I : Indexed file

Specify the record format using one of the following characters. However, when an indexed file is specified for the file organization, the record format cannot be specified as the attribute information in the file is used.

- f : Fixed-length
- v : Variable-length

- record-length : Specify the maximum record length when using variable-length records. However, when an indexed file is specified as the file organization, the record length cannot be specified as the attribute information in the file is used.

input-file-name

Specify the path name of a file to be sorted.

Example

To sort a relative file and output the result to a record sequential file:

```
$  cobfsort -ooutfile -s"(N,0,2),A;(X,10,5),D" -iR,v,80 relfile
```

Note

- Processing does not take place correctly if the record length of the input file is different from the actual file.
- Processing may not take place correctly if the file organization of the input file is different from the actual file.

19.3.6 Attribute

Displays attribute information (record length, record format, key information etc.) of an indexed file.

Specification Format

```
cobfattr  input-file-name
```

Parameter

input-file-name

Specify the path name of an indexed file for which attribute information is to be displayed.

Example

```
$  cobfattr ixdfile
```

Note

No attribute information of other than an indexed file is displayed.

19.3.7 Recovery

An indexed file may not be normally closed because of the abnormal end of a process. The 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
```

Parameter

recovery-file-name

Specify the path name of the indexed file to be recovered.

unrecoverable-data-file-name

Specify the path 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

```
$  cobfrcov ixdfile seqfile
```

Note

- When using the recovery function, a temporary work file (with name beginning "...UTY") of the same size as the recovery file is generated in /tmp.
- Processing takes place based on the file information retained in the file. Thus, if the information is corrupt, recovery cannot take place.
- The recovery function overwrites the file before recovery. Thus, in order to save the file before recovery, you need to have finished copying the file before the execution of recovery.
- This product does not require specifications for a high capacity file as is otherwise required by UNIX COBOL.
- When the File recovery command is executed at the same time for the same index file, operation is not guaranteed. Execute the second instance of the command after confirming execution completion of the first instance of the command.

19.3.8 Reorganization

Erases as much empty blocks in an indexed file as possible, and outputs the reorganized content to another indexed file. Erasing empty blocks results is a smaller file size.

Specification Format

```
cobfreog -ooutput-file-name  input-file-name
```

Parameter

output-file-name

Specify the path name of the indexed file (to be created) after reorganization.

input-file-name

Specify the path name of the indexed file to be reorganized.

Example

```
$  cobfreog -ooutfile ixdfile
```

Note

Erasing empty blocks may deteriorate file access performance.

Chapter 20 Remote Development Support Function

This chapter describes the remote distributed development support function.

20.1 Overview of Remote Development

20.1.1 What is Remote Development?

Using the remote development support functions, you can develop COBOL applications efficiently with widely spread Windows systems.

To perform remote development, you must have a system with NetCOBOL development system product including NetCOBOL Studio installed. In this chapter, the system where this version of NetCOBOL product is installed is called the "Server side" of remote development, and the system where NetCOBOL Studio is installed is called the "Client side" of remote development.

In remote development, the developer uses NetCOBOL Studio in client side system to perform development tasks. NetCOBOL Studio is connected to the server side system if required, development work like compilation etc. takes place at server side, and the result is displayed in NetCOBOL Studio.

20.1.2 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.
- As the systems are expensive, these must be shared among the developers.

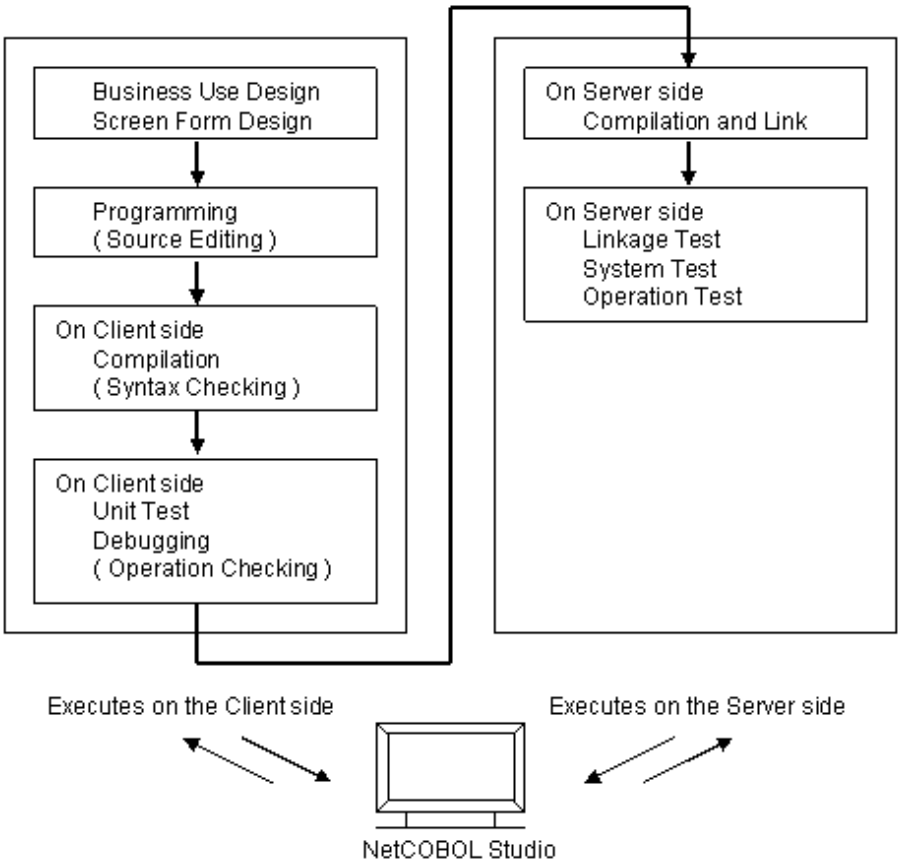
On the other hand, Windows system is easily available as a personal machine and developer can occupy its GUI-based environment exclusively.

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 expensive system at server side by performing development tasks such as source editing etc. on the client side as far as possible.

20.1.3 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/ remote debug functionality of NetCOBOL Studio.

20.1.4 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.

20.2 Remote Development Support Function

20.2.1 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 "20.3 NetCOBOL Remote Development Service" for details.

In remote development, a specified account is used to log in the server side system, and development tasks on the server side system are performed with that account.

Therefore, at the time of log-in using this account, the environment required for development must be configured. Refer to "3.1 Compilation and Link" for details about environment settings to compile and link COBOL programs.

Environment Variables to be set at sending resources

If code conversion needed for sending program resources is performed on the server side, the code conversion library can be changed.

COBOL_REMOTE_CONVERT_CHARACTER（Specify the code conversion library to transfer COBOL resources）

```
COBOL_REMOTE_CONVERT_CHARACTER = { ICONV | SYSTEM }
```

- ICONV

  The NetCOBOL runtime system converts the character-code (default).

- SYSTEM

  Character-code conversion is similarly carried out to V10 product by system API.

20.2.2 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" included in client-side products for details.

20.2.3 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.

20.3 NetCOBOL Remote Development Service

The NetCOBOL Remove Development Service (referred to as "Remote Development Service" hereinafter) must be run on the server side system while performing remote development from NetCOBOL Studio on the client side. The Remote Development Service accepts a

request from NetCOBOL Studio, logs in the server side system with the specified account and performs development related tasks on the server side system using that 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.

20.3.1 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.

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.

20.3.2 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 in the server side system with an administrative account and execute the following shell script:

```
/opt/FJSVXrds/bin/enable-rds.sh
```

This shell script starts the Remote Development Service and also changes the system so that the Remote Development Service is started automatically when the system starts.

The Remote Development Service uses port 61999 by default, to disclose its service. If port 61999 is already being used in the system, the port number must be changed. Refer to the description of port setting in "20.3.4 Configuring the Remote Development Service" for detail.

If 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 "20.3.1 Notes on security" before starting the Remote Development Service

Stopping the Remote Development Service

To stop the Remote Development Service, log in the server side system with an administrator account and execute the following shell script:

```
/opt/FJSVXrds/bin/disable-rds.sh
```

This shell script stops the Remote Development Service and also changes the system so that the Remote Development Service is not started automatically when the system starts.

Restore the changes that were made to firewall software or so if the Remote Development Service no longer needs to be exposed.

20.3.3 Log files of the Remote Development Service

This section explains the log files generated 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 login succeeded
- Connection end date and time

The commands executed by each user account after login are not recorded.

The time output to the log file is UTC (Universal Time, Coordinated).

The path of the Log file

The path of the log file is as follows:

```
/var/opt/FJSVXrds/log/rds.log
```

You can change the directory to output the log file by configuration of the Remote Development Server. Refer to the explanation for the logdir settings in "20.3.4 Configuring the Remote Development Service" for details.

You cannot change or delete log files unless you have administrative rights.

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 "20.3.4 Configuring the Remote Development Service" for details.

Backup files are located in the same directory as the log file output directory 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 "20.3.4 Configuring the Remote Development Service", for details.

20.3.4 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 directory, maximum size, and number of backup generations for log files

Change the configuration file entries for the settings that you want to change. The remote development service must then be restarted for the changes to take effect.

Configuration file of the Remote Development Service

The path of configuration file of the Remote Development Service is as follows:

```
/etc/opt/FJSVXrds/conf/rds.conf
```

This file is a text file and can be edited by text editors. However, you require administrative rights to change the configuration file.

Each setting is entered in a line in the configuration file. Enter the setting name, space, setting value, in the order it is stated. Lines starting with # are comments. The following is an example of configuration file contents:

```
# Sample configuration file
port 61999
```

```
logdir /var/opt/FJSVXrds/log
maxloggen 2
```

Refer to the list below for settings that can be entered in the configuration file.

Setting list of the Remote Development Service

The following table shows the settings that can be changed.

Setting name	Default value	Explanation
Port	61999	Specify the TCP/IP port number used by the Remote Development Service. Specify the port number in decimal form.
Logdir	/var/opt/FJSVXrds/log	Specify the directory path where remote development service log files are output.
maxlogsize	128	Specify the maximum size of the Remote Development Service log file. Specify the maximum size setting value in decimal form. Its unit is kilobytes. If 0 is specified, the Remote Development Service does not output log file.
maxloggen	2	Specify the number of generations of Remote Development Service log file backups to be retained. 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 regarded to be specified. If 0 is specified, no backup is retained.

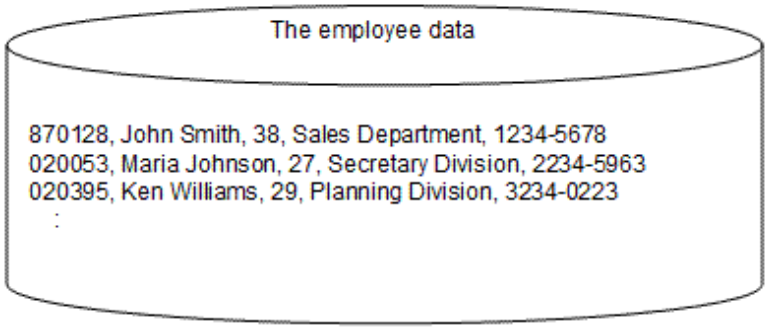
Chapter 21 Operation of CSV (Comma Separated Value) data

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.



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.
*>   :
     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.

- 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)

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.
*>   :
    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 "NetCOBOL 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.
*>  :
     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 "Table 21.1 CSV data generation errors" 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)

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 "NetCOBOL Language Reference".

 Note

The program does not operate correctly when the text file of TSV(Tab Separated Values) 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. For details, refer to "6.3 Using Line Sequential Files" and "6.8.1.2 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 "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.

```
  WORKING-STORAGE SECTION.
*>  :
  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 "Table 21.2 CSV data resolution errors" 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

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 "21.5.2 CBR_CSV_TYPE (Set the variation of generated CSV type)" of the STRING statement (format 2). MODE-1 is considered the default.

Example:

```
 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) "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.

21.5 Environment Variable

21.5.1 CBR_CSV_OVERFLOW_MESSAGE (Specify to suppress messages at CSV data operation)

```
$ CBR_CSV_OVERFLOW_MESSAGE=NO ; export CBR_CSV_OVERFLOW_MESSAGE
```

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

When characters other than "NO" are specified for the parameter, the output of messages is not controlled.

21.5.2 CBR_CSV_TYPE (Set the variation of generated CSV type)

$ CBR_CSV_TYPE= { MODE-1 / MODE-2 / MODE-3 / MODE-4 } ; export CBR_CSV_TYPE

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

This specification applies only to STRING statement without TYPE.

When the parameter is omitted, it is considered that MODE-1 is selected.

Refer to "21.4 Variation of CSV".

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, and then use the second section to obtain the correct format for the option.

A.1 List of Compiler Options

Options that relate to compile listings

- A.2.7 COPY (library text display)
- A.2.20 LINECOUNT (number of rows per page of the compile list)
- A.2.21 LINESIZE (number of characters per row in the compile list)
- A.2.22 LIST (determines whether to output the object program listings)
- A.2.24 MAP (whether a data map list, a program control information list, and a section size list are output)
- A.2.25 MESSAGE (whether the optional information list and statistical information list should be output for separately compiled programs)
- A.2.31 NUMBER (source program sequence number area specification)
- A.2.42 SOURCE (whether a source program listing should be output)
- A.2.52 XREF (whether a cross-reference list should be output)

Options that relate to compile messages

- A.2.6 CONF (whether messages should be output depending on the difference between standards)
- A.2.15 FLAG (Diagnostic Message Level)
- A.2.16 FLAGSW (whether pointer messages to language elements in COBOL syntax should be displayed)

Options that relate to source program interpretation

- A.2.1 ALPHAL (lowercase handling (in the program))
- A.2.4 BINARY (binary data item handling)
- A.2.10 CURRENCY (currency symbol handling)
- A.2.17 INITVALUE (handling an item having no VALUE clause in the working-storage section)
- A.2.19 LANGLVL (ANSI COBOL standard specification)
- A.2.28 NCW (Japanese-language character set specification for the user language)
- A.2.30 NSPCOMP (Japanese-language space comparison specification)
- A.2.34 QUOTE/APOST (figurative constant QUOTE handling)
- A.2.35 RCS (Handling national data items in a Unicode environment)
- A.2.36 RSV (the type of a reserved word)
- A.2.38 SCS(code system of source file)
- A.2.39 SDS (whether signs in signed decimal data items should be converted)
- A.2.40 SHREXT (determines how to treat the external attributes in a multithread program)
- A.2.43 SRF (reference format type)
- A.2.46 STD1 (alphanumeric character size specification)

- A.2.47 TAB (tab handling)
- A.2.53 ZWB (comparison between signed external decimal data items and alphanumeric data items)

Options that relate to source program analysis

- A.2.37 SAI (whether a source analysis information file is output)

Options that relate to object program generating

- A.2.2 ARITHMETIC (Specifies the operation mode)
- A.2.3 ASCOMP5 (Specifying an interpretation of binary items)
- A.2.9 CREATE (specifies a creating file)
- A.2.11 DLOAD (program structure specification)
- A.2.13 ENCODE (encoding form specification)
- A.2.18 LALIGN (Dealing data declaration of LINKAGE SECTION)
- A.2.23 MAIN (main program/sub-program specification)
- A.2.26 MODE (ACCEPT statement operation specification)
- A.2.27 NAME (whether object files should be output for each separately compiled program)
- A.2.29 NSP (handling of spaces related to national data item)
- A.2.32 OBJECT (whether an object program should be output)
- A.2.33 OPTIMIZE (global optimization handling)
- A.2.49 THREAD (specifies multithread program creation)

Options that relate to runtime control

- A.2.12 DNTB (Handling of blank trailing spaces in the DISPLAY-OF function and NATIONAL-OF function)
- A.2.14 EQUALS (same key data processing in SORT statements)
- A.2.51 TRUNC (Truncation Operation)

Options that relate to runtime resources

- A.2.41 SMSIZE (Memory capacity used by PowerBSORT)
- A.2.44 SSIN (ACCEPT statement data input destination)
- A.2.45 SSOUT (DISPLAY statement data output destination)

Options that relate to debugging functions

- A.2.5 CHECK (whether the CHECK function should be used)
- A.2.8 COUNT (whether the COUNT function should be used)
- A.2.48 TEST (whether the remote debug function of NetCOBOL Studio should be used)
- A.2.50 TRACE (whether the TRACE function should be used)

A.2 Compiler Option Specification Formats

Compiler Option Specification Methods and Their Priorities

Compiler options can be specified by:

1. Using the -WC option

2. Using a compiler directing statement within a source program (@OPTIONS) (see "2.3 Compiler Directing Statements").

Their priorities are in the order of (2) and (1). There are several methods for specifying option files and -WC options. Option files can be specified by:

1. Using the -i option in the environment variable "COBOLOPTS"
2. Using the -i option of the compile command

The specification priority is in the order of (2) and (1).

-WC option

The -WC option can be specified either by:

1. Using the environment variable "COBOLOPTS"
2. Using the compile command ()

The specification priority is in the order of (2) and (1). Of the -WC options specified, the one with the highest specification priority is handled as being effective. If multiple -WC options are specified, the option specified last will be effective.

Example

Example 1:

```
COBOLOPTS=-WC,"MAIN,MESSAGE";export COBOLOPTS
cobol -WC,"NOMAIN" a.cob
```

Here, the option *-WC,"MESSAGE"* and *-WC,"NOMAIN"* are effective *-WC,"MAIN"* is not.

Example 2:

```
cobol -WC,"NOMAIN" -WC,"MAIN,MESSAGE" a.cob
```

Here, the option *-WC, "MAIN,MESSAGE"* is effective, while the option *-WC,"NOMAIN"* is not.

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:

-WC

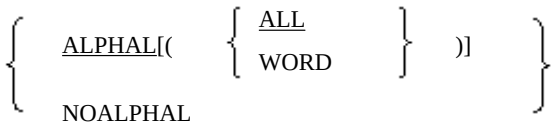
The compiler option can be specified in the-WC command option.

@

The compiler option can be specified in the compiler directing statement.

A.2.1 ALPHAL (lowercase handling (in the program))

-WC,@



This option specifies whether en-size lower-case letters and en-size upper-case letters in a source program are treated as equivalent (ALPHAL) or not (NOALPHAL).

Refer to "COBOL language" in "NetCOBOL Language Reference" for information concerning the COBOL language.

ALPHAL treats character constants as follows:

- ALPHAL(ALL)
  Lower-case letters and upper-case letters in the COBOL language are treated as being equivalent. Lower-case letters and upper-case letters in program name constants, and in the CALL statement, CANCEL statement, ENTRY statement, and INVOKE statement constants, are also treated as being equivalent.
- ALPHAL(WORD)
  Lower-case letters and upper-case letters in the COBOL language are treated as being equivalent. Lower-case letters and upper-case letters included in program name constants, and in the CALL statement, CANCEL statement, ENTRY statement, and INVOKE statement constants, are treated as being different.
- NOALPHAL
  Lower-case letters and upper-case letters in the COBOL language and constants are treated as being different.

Refer to "3.1.1.3 COB_COPYNAME(Specification of search condition of library text)", "Notes" in "3.1.3 How to Compile the Program Using the Library (COPY Statement)" and "8.3.6 Notes".

A.2.2 ARITHMETIC (Specifies the operation mode)

-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 arithmetic 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 V10.1.0 or earlier, or with other system.
- Refer to "1.7 Operation mode" in "NetCOBOL Language Reference" for details.

A.2.3 ASCOMP5 (Specifying an interpretation of binary items)

-WC,@

ASCOMP5({ NONE | ALL | BINARY | COMP })

This option specifies an interpretation of binary items.

- ASCOMP5(NONE)
  Interprets a binary item as declared.
- ASCOMP5(ALL)
  Assumes USAGE COMP-5 is specified for an item declared as USAGE BINARY, USAGE COMP, or USAGE COMPUTATIONAL.
- ASCOMP5(BINARY)
  Assumes USAGE COMP-5 is specified for an item declared as USAGE BINARY.
- ASCOMP5(COMP)
  Assumes USAGE COMP-5 is specified for an item declared as USAGE COMP or USAGE COMPUTATIONAL.

Note

Internal representation of data changes.

A.2.4 BINARY (binary data item handling)

-WC,@

BINARY({ WORD[, { MLBON / 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

- 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

Number of PIC Digits		Assigned Area Length	
Signed	Unsigned	BINARY(BYTE)	BINARY(WORD)
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)

-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 (NUMERIC)
  Checks for a data exception (whether a value in a numeric item conforms to the attribute format and whether the divisor is 0).
- 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(PRM)
  At the compile time, carry out the following investigations for data items described in USING phrase or RETURNING phrase of the CALL statement (except for the CALL identifier) that calls the internal program, and for data items described in USING phrase and RETURNING phrase 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 has can be determined at compile time.
  - If the data item is object reference, the class name specified in the USAGE OBJECT REFERENCE clause, FACTORY, and ONLY match.

  A data item specified in a USING or RETURNING phrase in the CALL statement that calls an external program and a data item specified in a USING or RETURNING phrase of the external program are checked to confirm the following during execution:
  - They have a matching number of parameters specified in the USING phrase and matching data item lengths.
  - However, if the difference between their numbers of parameters specified in the USING phrase is four or more, no error may be detected.
  - Their parameters specified in the RETURNING phrase have matching lengths.
  - If the RETURNING phrase is omitted, PROGRAM-STATUS is implicitly exchanged. A data item of 4 bytes is assumed to be specified.

  If lengths are judged during execution, the maximum value of the lengths specified during compilation is used for this check.

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, area destruction). 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 the debugging function terminates, specify NOCHECK, then recompile the program.
- In an arithmetic statement specifying an ON SIZE ERROR or a NOT ON SIZE ERROR phrase, a zero check is made for the divisor of ON SIZE ERROR, not for the divisor of CHECK (NUMERIC).
- A data exception check is made for CHECK (NUMERIC) when a zoned decimal item or packed decimal item is used for reference, and during a move from an alphanumeric data item or group item to a zoned decimal item or packed decimal item. However, it will not be a check target in the following cases:
  - Table elements which are specified ALL as the subscript.
  - Key items in the SEARCH ALL statement (this does not include subscript for key items that is 1 dimensional and is also a WHEN condition.)
  - 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
  - A move from an alphanumeric data item or group item to an object property of a zoned or packed decimal item.

Refer to "5.3 Using the CHECK Function".

A.2.6 CONF (whether messages should be output depending on the difference between standards)

-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.

The compiler options CONF(68) and CONF(74) are effective only if the compiler option LANGLVL(85) is specified.

Refer to "A.2.19 LANGLVL (ANSI COBOL standard specification)".

Information

CONF is effective when a program created according to the existing standard is changed to '85 ANSI COBOL.

A.2.7 COPY (library text display)

-WC,@

{ COPY / NOCOPY }

To display library text incorporated by the COPY statement in the source program listing, specify COPY; otherwise, specify NOCOPY.

Note

COPY is only effective when the compiler option SOURCE is specified.

Refer to "A.2.42 SOURCE (whether a source program listing should be output)".

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

-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.

Refer to "5.4 Using the COUNT Function".

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. Specify -c option with CREATE(REP).

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)

-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 "NetCOBOL Language Reference" for the currency symbols that can be used.

A.2.11 DLOAD (program structure specification)

-WC,@

```
{ DLOAD   }
{ NODLOAD }
```

Specify whether the program structure is a dynamic program structure (DLOAD) or not (NODLOAD).

Refer to "3.2.2 Linkage Type and Program Structure", "15.6.2 Compilation Processing in Dynamic Program Structures" and "8.1.2 Dynamic Program Structure".

A.2.12 DNTB (Handling of blank trailing spaces in the DISPLAY-OF function and NATIONAL-OF function)

-WC,@

```
{ DNTB   }
{ NODNTB }
```

For the data item specified in the argument-1 of NATIONAL-OF function and DISPLAY-OF function, specify whether blank trailing spaces are considered for conversion (DNTB) or are not considered (NODNTB).

A.2.13 ENCODE (encoding form specification)

-WC,@

```
ENCODE ( { SJIS } [, { SJIS                                  } ] )
         {      }     { { UTF16 } [, { LE } ] }
         { UTF8 }     { { 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.

Note

- If compile option RCS is specified, ENCODE option cannot be specified.
- If compile option ENCODE(SJIS,SJIS) is specified, please specify compile option SCS(SJIS) at the same time.
- If compile option ENCODE(SJIS,SJIS) is specified, the alphabet-name associated with following in the ALPHABET clause cannot be specified.
  - UTF8
  - UTF16
  - UTF16LE
  - UTF16BE
  - UTF32
  - UTF32LE
  - UTF32BE
- If compile option ENCODE(UTF8,UTF16) or ENCODE(UTF8,UTF32) is specified, the alphabet-name associated with the SJIS in the ALPHABET clause cannot be specified.
- If compile option ENCODE(UTF8,UTF16) is implicitly specified, the alphabet-name associated with the UTF32 in the ALPHABET clause cannot be specified.
- If the ENCODING clause is specified for the data item, specify the ENCODING clause is enabled.
- The alphabet-name associated with the UTF32 or UTF16 in the ALPHABET clause is subject to the specification of the endian of this option.
- The compile option ENCODE is effective only if the locale is Unicode. If the locale is other than Unicode, the compile option ENCODE does not have any meaning. Also, the alphabet-name associated with the encoding cannot be specified with the ENCODING clause.
- When described the report section, compile option ENCODE cannot be specified explicitly.

Point

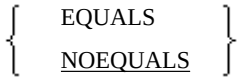
When compile option ENCODE (UTF8, UTF16) or ENCODE (UTF8, UTF32) is specified explicitly or implicitly, then spaces related to a national item become alphabetic character spaces. This handling can be done by specifying the compile option NSP.

See

"A.2.29 NSP (handling of spaces related to national data item)"

A.2.14 EQUALS (same key data processing in SORT statements)

-WC,@



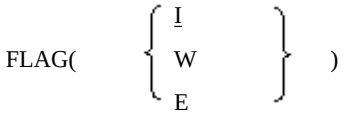
When records that have the same key are input to a SORT statement, specify EQUALS to ensure that the SORT output record order is identical to the input record order. Otherwise, specify NOEQUALS.

If NOEQUALS is specified, the sequence of records having the same key is not defined when the SORT statement outputs records.

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)

-WC,@



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.

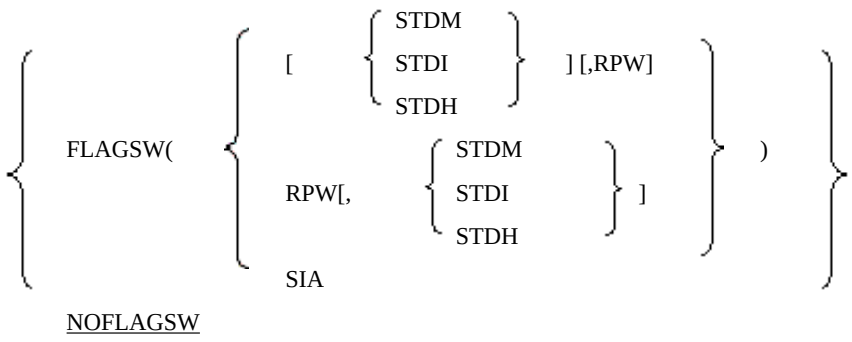
Note

The diagnostic message specified in the compiler option CONF is displayed regardless of the FLAG specification.

Refer to "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)

-WC,@



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 1985 ANS COBOL
- FLAGSW(STDI)
  Language elements that are not in the intermediate subset of the 1985 ANS COBOL
- FLAGSW(STDH)
  Language elements that are not in the high subset of the 1985 ANS COBOL
- FLAGSW(RPW)
  Language elements of the report writer function of the 1985 ANS COBOL
- FLAGSW(SIA)
  Language elements that are not in the range of Fujitsu System Integrated Architecture (SIA)

Information

- Use FLAGSW(SIA) to create a program that will be executed in another system.
- FLAGSW suboperands {STDM | STDI | STDH} and RPW cannot be omitted simultaneously.

A.2.17 INITVALUE (handling an item having no VALUE clause in the working-storage section)

-WC,@

{ INITVALUE(*xx*) | NOINITVALUE }

This option specifies whether an item having no VALUE clause in the working-storage section data is initialized (INITVALUE) with the specified value or not (NOINITVALUE).

xx specifies a 2-digit hexadecimal number. xx is required.

A.2.18 LALIGN (Dealing data declaration of LINKAGE SECTION)

-WC,@

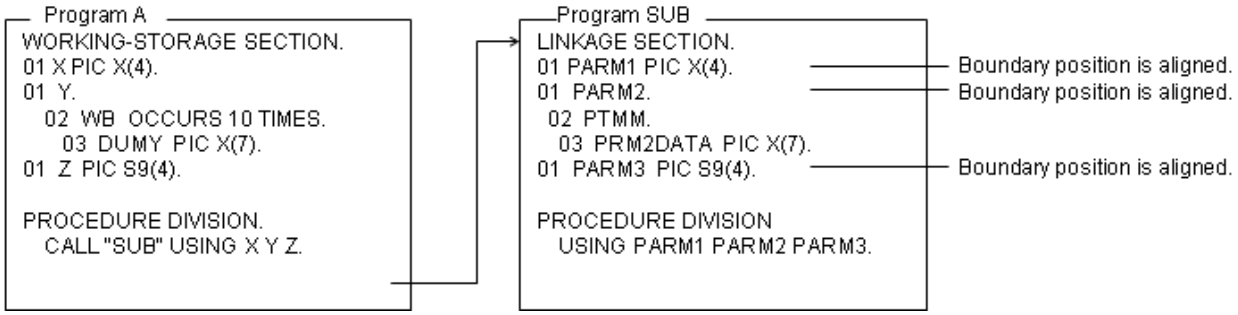
{ LALIGN | NOLALIGN }

If the object that is assumed to be aligned on the boundary of 8 bytes is generated, the data process speed improves.

When all data that corresponds to each data in LINKAGE SECTION is declared by 01 items in calling program, this option can be used.

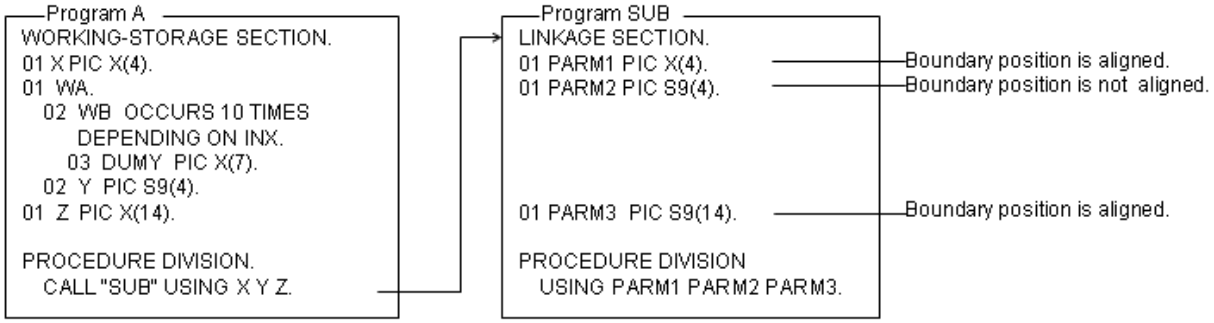
Example

Example 1: When aligned on the boundary (The performance improves if the option is specified)



The boundary position is aligned on the 8 byte boundary because parameter X, Y, and Z passed from program A to SUB are described in the head of WORKING-STORAGE SECTION and they are 01 items the boundary position.

Example 2: When not aligned on the boundary (The option cannot be specified)



The boundary position is fixed because Parameters X and Z passed from program A to SUB are described in the head of WORKING-STORAGE SECTION and they are 01 items. Y is not fixed because it is specified in the group item that contains variable items and the address is decided in the execution. Therefore, it is difficult for users to judge whether the boundary of data is aligned.

Note

If this option is specified, a source program containing data whose sorting boundary is not aligned can be compiled. However, the operation at execution time depends on the system used. (e.g., abnormal end, system error)

A.2.19 LANGLVL (ANSI COBOL standard specification)

-WC,@

FLAGLVL({ 85 | 74 | 68 })

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 LINECOUNT (number of rows per page of the compile list)

-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. If this option is not specified, LINECOUNT(0) is assumed.

Note

If a value 0 to 12 is specified, display without editing.

A.2.21 LINESIZE (number of characters per row in the compile list)

-WC,@

LINESIZE(*n*)

Specify the maximum number of characters (value resulting from conversion of alphanumeric characters displayed in the list) per line in compiler listings. The value can be 0, 80, or a 3-digit integer of 120 or more. When 0 is specified, the source program list is output all on the same line. If this option is not specified, LINESIZE(0) is assumed.

Note

- The option information list and diagnostic message list are output with a fixed number of characters (120) regardless of the maximum number of characters specified in LINESIZE.
- The maximum number of characters that can be specified is 136. If a value greater than 136 is specified in LINESIZE, 136 is used.

A.2.22 LIST (determines whether to output the object program listings)

-WC,@

{ LIST / NOLIST }

Specify whether to output (LIST) or not output (NOLIST) an object program listing. When LIST is specified, an object program listing is output to the file specified by compiler option -P. See "3.3.1.15 -P (Specification of the file name of the compile list)" and "3.3.1.10 -dp (Specification of compile list file directory)". Both "LIST" and "-P" options are necessary to output an object program listing.

A.2.23 MAIN (main program/sub-program specification)

-WC,@

{ 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.

- If you are compiling the COBOL programs by Continuous Compilation mode, or in a project, click on the [Main program] button.
- The MAIN option specified by the @OPTIONS compiler directing statement is only valid in the separately compiled program immediately after the compiler directing statement.

A.2.24 MAP (whether a data map list, a program control information list, and a section size list are output)

-WC,@

{ MAP / NOMAP }

This option specifies whether a data map list, a program control information list, and a section size list are output (MAP) or not (NOMAP). These lists are output to the file specified by the -P option. "3.3.1.15 -P (Specification of the file name of the compile list)" and "3.3.1.10 -dp (Specification of compile list file directory)".

Note

To output a data map list, a program control information list, and a section size list, the -P option must also be specified.

Refer to "3.3.1.15 -P (Specification of the file name of the compile list)"

A.2.25 MESSAGE (whether the optional information list and statistical information list should be output for separately compiled programs)

-WC,@

{ MESSAGE / NOMESSAGE }

To output an option information listing and to compile unit statistical information listing, specify MESSAGE; otherwise, specify NOMESSAGE.

A.2.26 MODE (ACCEPT statement operation specification)

-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.2.27 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.

If the NAME option is specified, the name of each object file is "each-separately compiled-program-ID.o".

This option is valid when the OBJECT option is specified. Whether -c option is specified or not, the cobol command outputs only object files and doesn't link them.

A.2.28 NCW (Japanese-language character set specification for the user language)

-WC,@

NCW({ STD / 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.

This option has no meaning if any other compile option other than SJIS is specified for SCS.

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.

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.2.29 NSP (handling of spaces related to national data item)

-WC,@

{ NSP / NONSP }

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.2.30 NSPCOMP (Japanese-language space comparison specification)

-WC,@

NSPCOMP ({ NSP | 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.
- In order to be handled/treated as new alphanumeric item a partially referred group item will not be considered/covered by NSPCOMP(ASP) option.

Information

In the code system (JEF) of an OSIV system, the value of a national space is equivalent to two characters of an ANK space. This characteristic is frequently used for nonnumeric comparisons. However, a national space does not have that equivalent value in SHIFT-JIS and EUC. Therefore, to move a COBOL program that operates on the OSIV system to this system, the source program must be modified.

If the NSPCOMP (ASP) compilation option is specified, the spaces being compared are treated as equivalent. Therefore, the operation in a comparison under the conditions described above can be expected to be the same as that on the system, even if the source program is not modified.

Each process that handles spaces as equivalent is performed according to stored character data. Therefore, if the operand is a group item, the process is not based on a subordinate item attribute. For example, if a subordinate alphanumeric data item contains data other than characters, a malfunction may occur.

A.2.31 NUMBER (source program sequence number area specification)

-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.

In this case, 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 COPY qualification-value.

- If NUMBER is specified, a sequence of identical consecutive values is not regarded as an error.
- If NUMBER is specified, the error jump feature in Builder cannot be used.
- 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 "SRF (reference format type)" in this Appendix.
- If NUMBER is specified, the gdb command cannot display the correct location of the source program during debugging.

A.2.32 OBJECT (whether an object program should be output)

-WC

{ OBJECT | NOOBJECT }

Specify OBJECT to generate an object program; otherwise, specify NOOBJECT. If an object program is output, the file is usually created in the directory of the source program. To create the file in another directory, specify the directory name.

Refer to "3.1.6 Files Used by the NetCOBOL Compiler".

A.2.33 OPTIMIZE (global optimization handling)

-WC,@

{ OPTIMIZE | NOOPTIMIZE }

Specify OPTIMIZE to generate a global-optimized object program; otherwise, specify NOOPTIMIZE.

The recommended specification for debugging is NOOPTIMIZE.

Refer to "Chapter 18 Using the Debugger", "Appendix C Global Optimization" and "A.2.48 TEST (whether the remote debug function of NetCOBOL Studio should be used)".

A.2.34 QUOTE/APOST (figurative constant QUOTE handling)

-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.

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.2.35 RCS (Handling national data items in a Unicode environment)

-WC

RCS({ UTF16 / UCS2 } [, { LE / BE }])

The encoding form of national data items in a Unicode environment is UTF-16. This option specifies whether big-endian or little-endian is used.

- RCS(UTF16,LE) or RCS(UCS2,LE)
  Little-endian is used in a Unicode environment.
- RCS(UTF16,BE) or RCS(UCS2,BE)
  Big-endian is used in a Unicode environment.

Refer to "Chapter 17 Unicode".

Note

- For V11 and later it is recommended to specify compile option ENCODE.
- When compile option RCS is specified, it is assumed that the compile option ENCODE is specified in the following way.
  - When RCS(SJIS) is specified, it is assumed ENCODE(SJIS,SJIS) is specified.
  - When RCS(UTF16) or RCS(UCS2) is specified, it is assumed ENCODE(UTF8,UTF16) is specified.

  However, alphabet-name associated with the UTF32 in the ALPHABET clause cannot be specified.
- When RCS(SJIS) is specified, it is assumed SCS(SJIS) is specified.

A.2.36 RSV (the type of a reserved word)

-WC,@

RSV ({ ALL / V111 / V112 / V122 / V125 / V30 / V40 / V61 / V70 / V81 / V90 / V1020 / V1050 })

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 V61
- 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(V1050)
  For Fujitsu NetCOBOL V10.3 and V10.4
- RSV(VSR2)
  For VS COBOL II REL2.0
- RSV(VSR3)
  For VS COBOL II REL3.0

A.2.37 SAI (whether a source analysis information file is output)

-WC,@

{ SAI / NOSAI }

This option specifies whether a source analysis information file is output (SAI) or not (NOSAI).

Refer to "3.1.6 Files Used by the NetCOBOL Compiler".

A.2.38 SCS(code system of source file)

-WC,@

SCS({ SJIS / UTF8 })

To set the code system of the COBOL source file and library to set the code system to Shift-JIS, specify SCS(SJIS). To set the code system to UTF-8, specify SCS(UTF8).

Note

ENCODE(SJIS[,SJIS]) must be used when SCS(SJIS) is specified.

A.2.39 SDS (whether signs in signed decimal data items should be converted)

-WC,@

{ SDS / NOSDS }

Specify this option to determine how to move a signed internal decimal data item to another signed internal decimal data item. Specify SDS to move the sign of the sending item as is; specify NOSDS to move a converted sign. X'B' and X'D' are treated as minus signs.

The other values are treated as plus signs. A plus sign of the sending item is converted to X'C'; a minus sign of the sending item is converted to X'D'.

A.2.40 SHREXT (determines how to treat the external attributes in a multithread program)

-WC,@

{ SHREXT / NOSHREXT }

Use this option when -Tm option or 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.2.41 SMSIZE (Memory capacity used by PowerBSORT)

-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 BSRTPRIM.memory_size of PowerBSORT. For details about which values are valid, refer to the PowerBSORT "Online Manual".

When this option is not specified, the size of the work area is automatically set by PowerBSORT. For details, 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() compilation 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.

A.2.42 SOURCE (whether a source program listing should be output)

-WC,@

{ SOURCE / NOSOURCE }

Specify SOURCE to output a source program listing; otherwise, specify NOSOURCE.

Refer to 3.3.1.15 -P (Specification of the file name of the compile list)" and "3.3.1.10 -dp (Specification of compile list file directory)".

Note

If the -P option is not specified, no source program list is output even if the SOURCE option is specified.

A.2.43 SRF (reference format type)

-WC

SRF({ FIX / FREE / VAR } [, { FIX / FREE / VAR }])

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 sued.

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.

A.2.44 SSIN (ACCEPT statement data input destination)

-WC,@

SSIN ({ *runtime-environment-variable-name* | SYSIN })

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)
  Standard input 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.

Refer to "9.1 ACCEPT/DISPLAY Function".

A.2.45 SSOUT (DISPLAY statement data output destination)

-WC,@

SSOUT ({ *runtime-environment-variable-name* | SYSOUT })

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)
  Standard output 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.

Refer to "9.1 ACCEPT/DISPLAY Function".

A.2.46 STD1 (alphanumeric character size specification)

-WC,@

STD1({ ASCII / JIS1 / JIS2 })

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.2.47 TAB (tab handling)

-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.2.48 TEST (whether the remote debug function of NetCOBOL Studio should be used)

-WC,@

{ TEST / NOTEST }

To use the remote debug function of NetCOBOL Studio, specify TEST; otherwise, specify NOTEST. If TEST is specified, the debugging information file used with the remote debug function of NetCOBOL Studio is created in the directory of the source program. To create the file in another directory, specify the directory name.

Note

If this option is specified with OPTIMIZE, the compiler does not perform global optimization. However, it is listed as OPTIMIZE in the established options.

Refer to "A.2.33 OPTIMIZE (global optimization handling)", "3.1.6 Files Used by the NetCOBOL Compiler", "3.3.1.7 -Dt (Specification for using the remote debug function of NetCOBOL Studio)" and "Chapter 18 Using the Debugger".

A.2.49 THREAD (specifies multithread program creation)

-WC,@

THREAD ({ MULTI / SINGLE })

Specify THREAD (MULTI) to generate a multithread object; otherwise, specify THREAD (SINGLE).

Note

IF the option THREAD (MULTI) is specified, it is necessary to link the object program as a multithreaded program. Therefore, specify -c or -Tm option additionally.

Refer to "Chapter 16 Multithread Programs".

A.2.50 TRACE (whether the TRACE function should be used)

-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 or -Dc. If they are specified together, only the latter is used. Refer to "COUNT" in this Appendix.

Refer to "5.2 Using the TRACE Function".

A.2.51 TRUNC (Truncation Operation)

-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.

If usage of PICTURE clause is used as described in the examples below, it will have the following effect:

- S999V9(integer part 3 digit) is moved to S99V99(integer part 2 digit): Digit dropped
- S9V999(integer part 1 digit) is moved to S99V99(integer part 2 digit): Digit not dropped

Note

- If NOTRUNC is specified and the number of digits in the integer part of the sending item is greater than that of the receiving item, the result is undefined.
- 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.

Refer to "A.2.33 OPTIMIZE (global optimization handling)".

A.2.52 XREF (whether a cross-reference list should be output)

-WC,@

{ XREF / NOXREF }

Specify whether a cross-reference list is output to a compile list (XREF) or not (NOXREF).

The cross-reference list shows user words and special registers in ascending order of character size. The cross-reference list is output to the file specified by -P option.

Refer to "3.3.1.15 -P (Specification of the file name of the compile list)" and "3.3.1.10 -dp (Specification of compile list file directory)".

Note

- If -P option is not specified, the cross-reference list is not output even though this option is specified.
- 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.2.53 ZWB (comparison between signed external decimal data items and alphanumeric data items)

-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 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

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
- FLAG
- LINECOUNT
- LINESIZE
- LIST
- MAP
- MESSAGE
- NUMBER
- OBJECT
- QUOTE/APOST
- SAI
- SOURCE

- SRF
- TEST
- TRACE
- XREF

Appendix B I-O Status List

Following table 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 an input-output statement is executed.

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.
Successful	04	--	The length of an input record is greater than the maximum record.
	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. - 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 with NO REWIND specified - 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.

Classification	I-O Status Value	Detailed Information	Meaning
			- 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. However, DUPLICATES is specified in the prime record key or alternate record key.
	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.
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.
	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

Classification	I-O Status Value	Detailed Information	Meaning
			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. - 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.

Classification	I-O Status Value	Detailed Information	Meaning
	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.
		ACM	A system error occurred.

Note

An input-output status value and detailed information are posted for a print file with the FORMAT clause and a representation file. A PowerFORM RTS notification code is posted for "FORM RTS" in detailed information. 0000 is posted for "--" in detailed information.

For details on PowerFORM RTS notification codes, see the *PowerFORM RTS Manual*.

Related manuals:

- FORM RTS online manual
- PowerFORM RTS online manual
- ACM Control Handbook

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  Subscript-1     PIC S99 BINARY.
  77  Subscript-2     PIC S99 BINARY.
  77  Subscript-3     PIC S99 BINARY.
  01  Group-item.
    02  Item-1 OCCURS 25 TIMES.
      03  Item-11  PIC XX OCCURS 10 TIMES.
    02  Item-2 OCCURS 35 TIMES.
      03  Item-21  PIC XX OCCURS 10 TIMES.
*>       :
PROCEDURE DIVISION.
*>       :
     MOVE SPACE TO  Item-11 (Subscript-1, Subscript-2).          *> [1]
*>       :
     MOVE SPACE TO  Item-21 (subscript-1, Subscript-3).          *> [2]
```

In example 1, if the value of "Subscript-1" remains unchanged between [1] and [2], the (Subscript-1 * 20) part becomes common from the "Item-1 (Subscript-1, Subscript-2)" address calculation expression "Item-1 - 22 + Subscript-1 * 20 + Subscript-2 * 2"(*1) and from the "Item-2 (Subscript-1, Subscript-3)" address calculation expression "Item-2 - 22 + Subscript-1 * 20 + Subscript-3 * 2." Therefore, [2] is optimized so that it uses the result of [1].

*1 : Item-1 + (Subscript-1 - 1) * 20 + (Subscript-2 - 1) * 2 = Item-1 - 22 + Subscript-1 * 20 + Subscript-2 * 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.
*>       :
PROCEDURE DIVISION.
```

```
*>       :
     COMPUTE Z1 = B1 * B2.  *> (1)
*>       :
     COMPUTE Z2 = B1 * B2.  *> (2)
```

In Example 2, B1 and B2 remain unchanged between (1) and (2), B1 * B2 becomes 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   Subscript               PIC S9(4)  BINARY.
  77   External-decimal-item   PIC S9(7)  DISPLAY.
  01   Group-item.
       02  Binary-item     PIC S9(7) BINARY  OCCURS 20 TIMES.
*>       :
PROCEDURE DIVISION.
*>       :
     MOVE 1  TO  Subscript.
 Start-of-loop.
     IF Binary-item (Subscript) = External-decimal-item GO TO End-of-loop.
*>       :
     ADD 1 TO  Subscript.
     IF  Subscript IS <= 20 GO TO Start-of-loop.
 End-of-loop.
```

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 only 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   Subscript          PIC S9(4) COMP-5.
  01   Group-item.
       02  Repeating-item PIC X(10) OCCURS 20 TIMES.
*>       :
PROCEDURE DIVISION.
*>       :
 Start-of-loop.
     IF  Repeating-item (Subscript) =  ...
*>       :
     ADD 1 TO  Subscript.                                    *> [1]
     IF  Subscript IS <= 20 GO TO Start-of-loop.             *> [2]
```

Subscript is a guidance variable(*1). Here, a new guidance variable (t) is introduced, and the (Subscript * 10) multiplication in the repeating-item (Subscript) address calculation expression "Repeating-item - 10 + Subscript * 10"(*2) is replaced with t. "ADD 10 TO t" is generated after [1]. Moreover, if Subscript is not used with another element in the loop and the Subscript value calculated in the loop is not used after control exits from the loop, [2] is replaced with "IF t IS <= 200 GO TO Start-of-loop." and [1] is deleted.

*1 : Defined only recurrently by the constant in the loop.

*2 : Repeating-item + (Subscript - 1) * 10 = Repeating-item - 10 + Subscript * 10

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).
*>       :
PROCEDURE DIVISION.
*>       :
     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.2.33 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 the global optimization

The compiler does not perform global optimization on the following programs:

- Programs that do not define the items and index names having the attributes to target by global optimization
- Programs using the segmentation function (segmentation module)

- Programs specifying the compiler option TEST. For details, refer to "A.2.48 TEST (whether the remote debug function of NetCOBOL Studio should be used)".

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
- Program using no numeric data items but only alphanumeric data items
- Programs referencing no declaratives from declaratives
- Programs referencing declaratives from no declaratives
- Programs specifying the compiler option TRUNC. For details, refer to "A.2.51 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 to 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.2.51 TRUNC (Truncation Operation)".

Appendix D Intrinsic Function List

This appendix explains the intrinsic functions provided by the NetCOBOL compiler.

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.3 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 sending side of the MOVE statement.

The following example uses an integer function in the COMPUTE statement.

Example

[COBOL program]

```
 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 sending side of the MOVE statement.

Alphanumeric Function

An alphanumeric function can be written anywhere an alphanumeric data item can be written.

Example

[COBOL program]

```
DATA            DIVISION.
WORKING-STORAGE SECTION.
01 LOWCASE      PIC X(8) VALUE "netcobol".
PROCEDURE DIVISION.
    DISPLAY "Before conversion: " LOWCASE.
    DISPLAY "After conversion: " FUNCTION UPPER-CASE(LOWCASE).
```

[Execution result]

```
Before conversion: netcobol
After conversion: NETCOBOL
```

In this example, the DISPLAY statement is used to display characters converted into upper case. However, the converted characters can be written in the sending 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

[COBOL program]

```
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
```

A conversion mode of the NATIONAL function can be specified with the runtime environment variable CBR_FUNCTION_NATIONAL.

Pointer Data Function

For information on the coding the pointer data function, see "11.3 Using the ADDR and LENG Functions" in Chapter 12.

D.2 Function Type Determined by Argument Type

Some function types are determined by the type of 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.

The TZ environment variable is required in order to use the CURRENT-DATE function. For detailed information about the TZ environment variable, execute the following:

```
$ man tzset
```

Example [sh]

```
$ TZ="PST8PDT" ; export TZ
```

Example [csh]

```
$ setenv TZ "PST8PDT"
```

If any date is specified in the CBR_JOBDATE environment variable, a date specified with the CURRENT-DATE function can be received.

Example

```
DATA            DIVISION.
 WORKING-STORAGE SECTION.
  01 TODAY.
   02 T-YEAR   PIC X(4).
   02 T-MONTH  PIC X(2).
   02 T-DAY    PIC X(2).
   02        PIC X(13).
 PROCEDURE DIVISION.
     MOVE FUNCTION CURRENT-DATE TO TODAY.
```

1999.09.01 is set in the CBR_JOBDATE environment variable at program execution time.

After the move operation, 1999 is saved in the T-YEAR variable, 09 in the T-MONTH variable, and 01 in the T-DAY variable.

For the environment variable specification format, see "9.1.8 Entering any date".

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.".
```

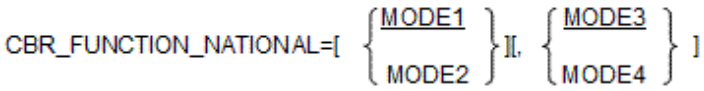
[Execution result] (Suppose "19920521" is input for the reference date.)

```
6385 days have passed from the deposit date(1992/05/21).
The total interest as of 2009/11/13 is  42,522 yen.
```

D.5 Conversion Mode of the NATIONAL Function

The conversion mode of the NATIONAL function is specified for the CBR_FUNCTION_NATIONAL environment variable.

D.5.1 CBR_FUNCTION_NATIONAL (specification of the conversion mode of the NATIONAL function)



Either MODE1 for conversion as usual in V40 or earlier, or MODE2 for conversion with a visual approximation, is specified in the first operand for the conversion mode of the NATIONAL function. If the operand is omitted, MODE1 is assumed to be specified.

If the character set is Unicode, either MODE3 (conversion such as for UNIX system) or MODE4 (conversion such as for Windows system) is specified in the second operand. If this operand is omitted, MODE3 is assumed to be specified.

Note

- If a specification contains an error, the entire specification is assumed to be omitted.

If neither MODE3 nor MODE4 is specified, the system behavior depends on the platform.

The following tables list differences between MODE1 and MODE2.

Table D.1 In Unicode

Specification	Conversion rule
MODE1	"-" (0xB0) is converted to "-" (0x1520). "`" (0x60) is converted to "" (0x1820).
MODE2	"-" (0xB0) is converted to "-" (0xFC30). "`" (0x60) is converted to "`" (0x40FF).

The following table lists differences between MODE3 and MODE4.

Table D.2 In Unicode

Specification	Conversion rule
MODE3	"-" (0x2D) is converted to "-" (0x1222: MINUS SIGN). "~" (0x7E) is converted to "~" (0x1C30: WAVE DASH).
MODE4	"-" (0x2D) is converted to "-" (0x0DFF: FULLWIDTH HYPHEN-MINUS). "~" (0x7E) is converted to "~" (0x5EFF: FULLWIDTH TILDE).

D.6 Intrinsic Function List

Following table lists the intrinsic functions.

Table D.3 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 characters into Alphanumeric characters.	Alphanumeric
	NATIONAL-OF	Converts 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

Classification	Function	Explanation	Function type
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
	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

Classification	Function	Explanation	Function type
	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 Environment Variable List

The table below shows the environment variables required to execute COBOL.

The meanings of the symbols used in the Conditions column are as follows:

A

Indicates that an environment variable needs to be set when using the product under specified conditions. In the runtime system, this is available when the runtime initialization file is set.

B

Indicates that an environment variable needs to be set when using the product under specified conditions. In the runtime system, this is not available if the runtime initialization file is set.

-

Indicates no environment variables are needed.

Table E.1 Environment variable list to execute COBOL

Environment Variable Name	Content (Value) Notes	Conditions	
BSORT_TMPDIR	Path name of the directory where work files are created Used for specifying the directory of work files used with the SORT/MERGE statement (See "Chapter 10 Using SORT/MERGE Statements (Sort-Merge Function)").	Compiler Runtime system Remote debug function File Utility	- A - B
CBR_ATTACH_TOOL	The connection destination information and additional path list that invokes the remote debug function of NetCOBOL Studio from a debug program When invoking the remote debug function of NetCOBOL Studio from a debug program.	Compiler Runtime system Remote debug function File Utility	- - A -
CBR_CBRFILE	Runtime environment file name When using the runtime initialization file (See "Chapter 4 Executing Programs").	Compiler Runtime system Remote debug function File Utility	- B - -
CBR_CBRINFO	Character-string : YES (if specified) When outputting the COBOL program runtime information as the runtime message (JMP0070I-I) (see "Runtime Messages" in Appendix N). If the string YES is specified, execution-time information on the program is output as an execution-time message (JMP0070I-I). If the string ENV is specified, execution-time information on a program is output as an execution-time message (JMP0070I-I) and execution-time environment variable information is output to the file specified in the CBR_MESSOUTFILE environment variable. When specifying the string ENV, also set the CBR_MESSOUTFILE environment variable. If CBR_MESSOUTFILE is omitted, execution-time environment variable information is not output.	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_CLASSINFFILE	Path name of the class information file	Compiler	-

Environment Variable Name	Content (Value) Notes	Conditions	
	When changing the class information to be used in an object-oriented program (See "Chapter 15 Developing and Executing Object-Oriented Programs".	Runtime system Remote debug function File Utility	A - -
CBR_CLOSE_SYNC	Character-string : YES (if specified) When immediately outputting data at execution of the CLOSE statement.	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_COMPOSER_CONSOLE	Management name defined by log definition file of Interstage Business Application Server When the DISPLAY UPON CONSOLE output destination is set to the Interstage Business Application Server general log (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_COMPOSER_MESS	Management name defined by log definition file of Interstage Business Application Serve When the output destination of execution-time messages is set to the Interstage Business Application Server general log (See "4.3.4 Outputting execution time messages to the Interstage Business Application Server general log").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_COMPOSER_SYSERR	Management name defined by log definition file of Interstage Business Application Server When the DISPLAY UPON SYSERR output destination is set to the Interstage Business Application Server general log (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_COMPOSER_SYSOUT	Management name defined by log definition file of Interstage Business Application Serve When the DISPLAY UPON SYSOUT output destination is set to the Interstage Business Application Server general log (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_CONVERT_CHARACTER	Character-string : ICONV(NetCOBOL runtime system) or SYSTEM(System API) This variable specifies the code conversion library to use when the runtime system converts a character-code.	Compiler Runtime system Remote debug function File Utility	- A A -
CBR_CSV_OVERFLOW_MESSAGE	Character-string : NO (when specified) Used for suppressing messages JMP0262I-W and JMP0263I-W upon execution of STRING statement (format 2) or UNSTRING statement (format 2) (See "Chapter 21 Operation of CSV (Comma Separated Value) data").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_CSV_TYPE	Character-string : MODE-1, MODE-2, MODE-3 or MODE-4	Compiler Runtime system Remote debug function	- A -

Environment Variable Name	Content (Value) Notes	Conditions	
	Used for setting the variation of generated CSV type (See "Chapter 21 Operation of CSV (Comma Separated Value) data").	File Utility	-
CBR_DISPLAY_CONSOLE_OUTPUT	Character-string : SYSLOG (when specified) When the DISPLAY UPON CONSOLE output destination is set to Syslog (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_SYSOUT_OUTPUT	Character-string : SYSLOG (when specified) When the DISPLAY UPON SYSOUT output destination is set to Syslog (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_SYSERR_OUTPUT	Character-string : SYSLOG (when specified) When the DISPLAY UPON SYSERR output destination is set to Syslog (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_CONSOLE_SYSLOG_LEVEL	Character-string : I, W or E When the logging level to use for the DISPLAY UPON CONSOLE output to Syslog is changed (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_SYSOUT_SYSLOG_LEVEL	Character-string : I, W or E When the logging level to use for the DISPLAY UPON SYSOUT output to Syslog is changed (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_SYSERR_SYSLOG_LEVEL	Character-string : I, W or E When the logging level to use for the DISPLAY UPON SYSERR output to Syslog is changed (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_CONSOLE_SYSLOG_IDENT	Identity name (when specified) When the Identity Name to use for the DISPLAY UPON CONSOLE output to Syslog is changed (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_SYSOUT_SYSLOG_IDENT	Identity name (when specified) When the Identity Name to use for the DISPLAY UPON SYSOUT output to Syslog is changed (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_DISPLAY_SYSERR_SYSLOG_IDENT	Identity name (when specified) When the Identity Name to use for the DISPLAY UPON SYSERR output to Syslog is changed (See	Compiler Runtime system Remote debug function	- A -

Environment Variable Name	Content (Value) Notes	Conditions	
	"Chapter 9 Using ACCEPT and DISPLAY Statements").	File Utility	-
CBR_ENTRYFILE	Entry information file name Used for calling a subprogram in the dynamic program structure	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_EXFH_API	Entry-name of External file handler When an External file handler is to be used, an entry-name for it must be specified (See "6.8.8 External file handler").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_EXFH_LOAD	Shared object file name of External file handler When an External file handler is to be used, a shared object file name for it must be specified (See "6.8.8 External file handler").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_FCB_NAME	FCB name If changing the FCB default value in print files that form descriptors are used, set FCB name used for default in it (See "Chapter 7 Printing").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_FILE_BOM_READ	Character-string : CHECK, DATA, or AUTO Specify the treatment of the BOM when referencing a Unicode file (See "6.3 Using Line Sequential Files").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_FILE_SEQUENTIAL_ACCESS	Character-string : BSAM The batch specification for High-speed file processing (See "Batch specification").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_FILE_USE_MESSAGE	Character-string : YES (when specified) Used with a valid error handling procedure for output of an execution-time message of an input-output error (See "6.6 Input-Output Error Processing").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_FUNCTION_NATIONAL	Character-string : MODE1 (conversion as V40L20 or before compatible) (default value) or MODE2 (more visual conversion). Character-string : MODE3 (conversion such as for UNIX system) or MODE4 (conversion such as for Windows system) When specifying the conversion mode of the NATIONAL function (See "Appendix D Intrinsic Function List").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_INPUT_BUFFERING	Character-string : YES	Compiler	-

Environment Variable Name	Content (Value) Notes	Conditions	
	When specifying the head-ahead processing of file input (See "Chapter 6 File Processing").	Runtime system Remote debug function File Utility	A - -
CBR_INSTANCEBLOCK	Character-string : USE (blocking) or UNUSE (no blocking) When specifying how to acquire an object instance in an object-oriented program.	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_JOBDATE	Optional date (YY.MM.DD or YYYY.MM.DD (when specified) Used with ACCEPT to FROM DATE or the built-in CURRENT-DATE function to obtain any date (See "9.1.8 Entering any date" and "D.3 Obtaining the Year Using the CURRENT-DATE Function").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_LP_OPTION	Option character of the lp command When the ASSIGN clause is PRINTER in the file control entry (See "Chapter 7 Printing").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_MEMORY_CHECK	Character-string : MODE1 Used with the memory check function to check a runtime system area when an application is executed (See "5.5 Using the Memory Check Function").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_MESS_LEVEL_CONSOLE	Character-string : NO or I or W or E or U Used for changing the severity level of an execution-time message to be displayed (output to a file if CBR_MESSOUTFILE is specified) (See "4.3.1 Specifying execution time message severity levels").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_MESS_LEVEL_SYSLOG	Character-string : NO or I or W or E or U Used for changing the severity level of an execution-time message to be output to Syslog (See "4.3.3 Outputting execution time messages to Syslog").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_MESSOUTFILE	Name of the file to which execution-time messages are output Used for output of an execution-time message and the results of the DISPLAY statement associated with SYSERR to a file (See "4.3.2 Outputting Error Messages to Files").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_PRINTFONTTABLE	Font table common to the same runtime environment When using the font table in a print file (See "7.1.9 Font Tables").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_PRT_INF	Path name of the print information file	Compiler	-

Environment Variable Name	Content (Value) Notes	Conditions	
	When using a print information file (See "Chapter 7 Printing").	Runtime system Remote debug function File Utility	A - -
CBR_SSIN_FILE	Character-string : THREAD (if specified) Specify to open the input file of each thread.	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_SYMFOWARE_THREAD	Character-string : MULTI (if specified) When enabling operation of a multithread program linked to SymfoWARE.	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_SYSERR_EXTEND	Character-string : YES (if specified) When appending the process ID and thread ID information to the DISPLAY statement output to SYSERR.	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_THREAD_TIMEOUT	wait time (second) When changing the wait time if an infinite wait is specified for the thread synchronization control subroutine.	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_TRACE_FILE	Trace information path name When using the TRACE function (See "Chapter 5 Debugging Programs").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_TRACE_PROCESS_MODE	Character-string : MULTI (if specified) When using the TRACE function (See "Chapter 5 Debugging Programs").	Compiler Runtime system Remote debug function File Utility	- A - -
CBR_TRAILING_BLANK_RECORD	Character-string : REMOVE (removing the trailing blank in the record) or VALID (enable the trailing blank) When executing the WRITE statement of the line sequential file (See "6.8.2.2 Specification of Trailing Blanks in the Line Sequential File").	Compiler Runtime system Remote debug function File Utility	- A - -
COBOL_REMOTE_CONVERT_CHARACTER	Character-string : ICONV(NetCOBOL remote development system) or SYSTEM(System API) Specify the code conversion library to transfer COBOL resources	Compiler Runtime system Remote development Remote debug function File Utility	- - A - -
COBCOPY	Library test storage directory	Compiler	

Environment Variable Name	Content (Value) Notes	Conditions	
		Runtime system Remote debug function File Utility	
COB_COPYNAME	Search criteria for the library text When specifying the search criteria for the library text (See "Chapter 3 Compiling and Linking Programs").	Compiler Runtime system Remote debug function File Utility	B - B -
COB_LIBSUFFIX	Extension of a library file When changing the extension of the library file.	Compiler Runtime system Remote debug function File Utility	B - - -
COBOLOPTS	List of the options specified with a cobol command When specifying the default value of a compile option at compilation with a cobol command (See "Chapter 3 Compiling and Linking Programs").	Compiler Runtime system Remote debug function File Utility	B - - -
COB_REPIN	Input directory of a repository file When specifying the default directory to which the repository file is input (See "Chapter 3 Compiling and Linking Programs").	Compiler Runtime system Remote debug function File Utility	B - - -
FCBDIR	Path name to the directory in which the FCB module is stored. When using FCB (See "Chapter 7 Printing").	Compiler Runtime system Remote debug function File Utility	- A - -
FORMLIB	Name of the directory storing the form descriptor files. When using the form descriptor files (See "Chapter 3 Compiling and Linking Programs").	Compiler Runtime system Remote debug function File Utility	B - - -
FOVLDIR	Path name to the directory in which the Form Overlay Patterns is stored. When using Form Overlay Patterns (See " Chapter 7 Printing").	Compiler Runtime system Remote debug function File Utility	- A - -
GOPT	List of the runtime options When specifying runtime options (See "4.2.2 Specifying Runtime Options").	Compiler Runtime system Remote debug function File Utility	- A - -
LANG	Language-name Set the codes that use COBOL.	Compiler Runtime system Remote debug function	B A B

Environment Variable Name	Content (Value) Notes	Conditions	
		File Utility	B
LC_ALL	Language-name	Compiler	B
	Set the same value as the one in LANG if the setting is required.	Runtime system	A
		Remote debug function	B
		File Utility	B
LD_LIBRARY_PATH	- Path to the library to be linked - Path to the module invoked from the dynamic link or a dynamic program.	Compiler	B
		Runtime system	B
		Remote debug function	B
	Required. Refer to the template to set the environment variable under the directory of /opt/FJSVcbl64/config for values to set.	File Utility	B
MANPATH	Path name of the online manual	Compiler	B
	Required	Runtime system	B
		Remote debug function	-
		File Utility	B
MEFTDIR	Path name of the directory in which the information file used by connected products is stored.	Compiler	-
	When using PowerFORM RTS (See "Chapter 7 Printing").	Runtime system	A
		Remote debug function	-
		File Utility	-
MGPRM	Runtime parameter of OSIV system form	Compiler	-
	When specifying runtime parameter of OSIV system form	Runtime system	A
		Remote debug function	-
		File Utility	-
NLSPATH	Path to the online help and message catalog	Compiler	B
	Required	Runtime system	B
		Remote debug function	-
		File Utility	B
PATH	Command search path	Compiler	B
	Required	Runtime system	B
		Remote debug function	B
		File Utility	B
SMED_SUFFIX	Extension of form descriptor files	Compiler	B
		Runtime system	-
		Remote debug function	B
		File Utility	-
SYSCOUNT	Path name of the COUNT information file	Compiler	-
	When using the COUNT function (See "Chapter 5 Debugging Programs").	Runtime system	A

Environment Variable Name	Content (Value) Notes	Conditions	
		Remote debug function File Utility	-
TMPDIR	Temporary storage of a work file When using the Recovery command of the COBOL FILE UTILITY (See "Chapter 19 COBOL File Utility").	Compiler Runtime system Remote debug function File Utility	- B - B
Library name	Library test storage directory When describing the library name in the COPY statement (See "Chapter 3 Compiling and Linking Programs").	Compiler Runtime system Remote debug function File Utility	B - B -
File-identifier	- Path name of the file - Path name of the information file used by connected products - Connected product identifier When performing file processing When using a print file (See "Chapter 6 File Processing", "Chapter 7 Printing" and "Chapter 10 Using SORT/MERGE Statements (Sort-Merge Function)").	Compiler Runtime system Remote debug function File Utility	- A - -
Name specified in compiler option SSIN	Path name of a file When specifying the input file in the ACCEPT statement with the compiler option SSIN (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -
Name specified in compiler option SSOUT	Path name of a file When specifying the output file in the DISPLAY statement with the compiler option SSOUT (See "Chapter 9 Using ACCEPT and DISPLAY Statements").	Compiler Runtime system Remote debug function File Utility	- A - -

See the template for setting environment variables under the directory /opt/FJSVcbl64/config for the values to be set in the following environment variables:

- PATH
- NLSPATH
- MANPATH

Appendix F Writing Special Literals

This appendix explains how to write literals for specifying names (for example, program and file names) defined in the system.

F.1 Program-Name Literal

The length of the program name literal must not exceed 60 bytes.

F.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:

```
"File-name"
```

You must comply with the COBOL user word rules when you specify the file name.

F.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 format:

```
"[path-name][file-reference-name]"
```

You can enter either an absolute path or a relative path for the path-name. If the path-name is omitted, the file specified in the file-reference-name will be considered in the current directory.

F.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.  SP-PROC  AS  "SP-CLASS-001".
```

Method name:

```
 METHOD-ID.  GET-VAL  AS  "VALUE".
```

The following characters cannot be used for a literal specified in the AS phrase. The user must confirm that linker rules are observed.

- Character strings that begin with an underscore (_).

Appendix G Subroutines Offered by NetCOBOL

This appendix explains the NetCOBOL subroutines.

G.1 Subroutines for Obtaining System Information

COBOL offers the subroutines below for obtaining system information:

- Process ID
- Thread ID

You can use these subroutines with COBOL programs. The following section explains how to fetch the system information that calls the subroutines from COBOL programs.

Note

To call a subroutine got by the system information using a dynamic program structure, you must use the following entry information file. For details about entry information, refer to "4.1.3 Subprogram entry information".

```
[ENTRY]
Subroutine-name=librcobol.so
```

G.1.1 The Subroutine for Obtaining a Process ID

You can use the COB_GET_PROCESSID subroutine to get the process ID of the process that calls this subroutine.

Call Format

```
CALL  "COB_GET_PROCESSID" USING BY REFERENCE data-name-1.
```

Explanation of Parameter

Data-name-1

```
01 data-name-1   PIC  9(9) COMP-5.
```

Specify the area for the process ID that is notified by the subroutine.

Note

When COB_GET_PROCESSID is called, special register PROGRAM-STATUS is updated with an irregular value. To avoid this, describe dummy data item (PIC S9(9) COMP-5) in the RETURNING phrase of the CALL statement.

G.1.2 The Subroutine for Obtaining a Thread ID

You can use the COB_GET_THREADID subroutine to get the thread ID of the thread that calls this subroutine.

Call format

```
CALL  "COB_GET_THREADID" USING  BY  REFERENCE data-name-1.
```

Explanation of Parameter

Data-name-1

```
01 data-name-1   PIC  9(9) COMP-5.
```

Specify the area for the thread ID that is notified by the subroutine.

Note

When COB_GET_THREADID is called, special register PROGRAM-STATUS is updated with an irregular value. To avoid this, describe dummy data item (PIC S9(9) COMP-5) in the RETURNING phrase of the CALL statement.

G.2 Subroutines Used in Linkage with another Language

G.2.1 The 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.

Specification Method

Data definition (calling with C language):

```
   Type declaration part:
     extern void JMPCINT2 (void);

   Procedure part:
     JMPCINT2();
```

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 start of a run unit, refer to "8.1.1 COBOL Inter-Language Environment" and "16.3.1 Runtime Environment and Run Unit".

Example

```
extern void JMPCINT2(void);
extern void JMPCINT3(void);
extern int  COBSUB1(void);        // COBOL program
extern int  COBSUB2(void);        // COBOL program

int csub(void) {
    JMPCINT2();                   // Starting a run unit
    COBSUB1();                    // COBOL program
    COBSUB2();                    // COBOL program
    JMPCINT3();                   // Closing the run unit
}
```

G.2.2 The Run Unit End Subroutine

After a run unit is started by the runtime start subroutine, the run unit end subroutine is used from another language program to close the run unit.

Specification Method

Data definition (calling with C language):

```
Type declaration part:
  extern void JMPCINT3 (void);

Procedure part:
  JMPCINT3();
```

Interface

No parameter is required for calling.

Return Code

The subroutine posts no return code.

Notes

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, refer to "8.1.1 COBOL Inter-Language Environment" and "16.3.1 Runtime Environment and Run Unit".

Example

```
extern void JMPCINT2(void);
extern void JMPCINT3(void);
extern int  COBSUB1(void);                  // COBOL program
extern int  COBSUB2(void);                  // COBOL program

int csub(void) {
    JMPCINT2();                             // Starting a run unit
    COBSUB1();                              // COBOL program
    COBSUB2();                              // COBOL program
    JMPCINT3();                             // Closing the run unit
}
```

G.2.3 The 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.

Specification Method

Data definition (calling with C language):

```
Type declaration part:
  extern void JMPCINT4(void);

Procedure part:
  JMPCINT4();
```

Interface

No parameter is required for calling.

Return Code

The subroutine posts no return code.

Notes

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 information on closing a run unit, refer to "8.1.1 COBOL Inter-Language Environment" and "16.3.1 Runtime Environment and Run Unit".

- In a process model 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 process model program, the subroutine returns without responding.

Example

```
#include <pthread.h>
extern int COBSUB(void);
extern void JMPCINT4(void);
int main(void){
   for (i=0;i<THREADNUM;i++) {
      pthread_create(&tid,NULL,
                     (void *)COBSUB,NULL);
   }
   pthread_join(tid,NULL);
   JMPCINT4();
     :
   The processing except other COBOLs.
     :
    return 0;
}
```

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.     COBSUB.
 PROCEDURE  DIVISION.
    DISPLAY "HELLO!".
    EXIT PROGRAM.
```

Close the run-time environment
(Release the COBOL resource)

G.3 File Access Routines

The Application Program Interface (API) function group is offered to enable you to organize access to COBOL files using C language. You can use COBOL file access routines to perform the following tasks using C language:

- Input-output from existing resources to write and overwrite files created using COBOL applications.
- Create organization files in COBOL format.
- Share and lock COBOL applications and files.
- Analyze the file attributes and record key configuration for existing indexed files.

For details about file access routines, refer to the README stored in the product, or the "NetCOBOL File Access Routine User's Guide".

G.4 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:

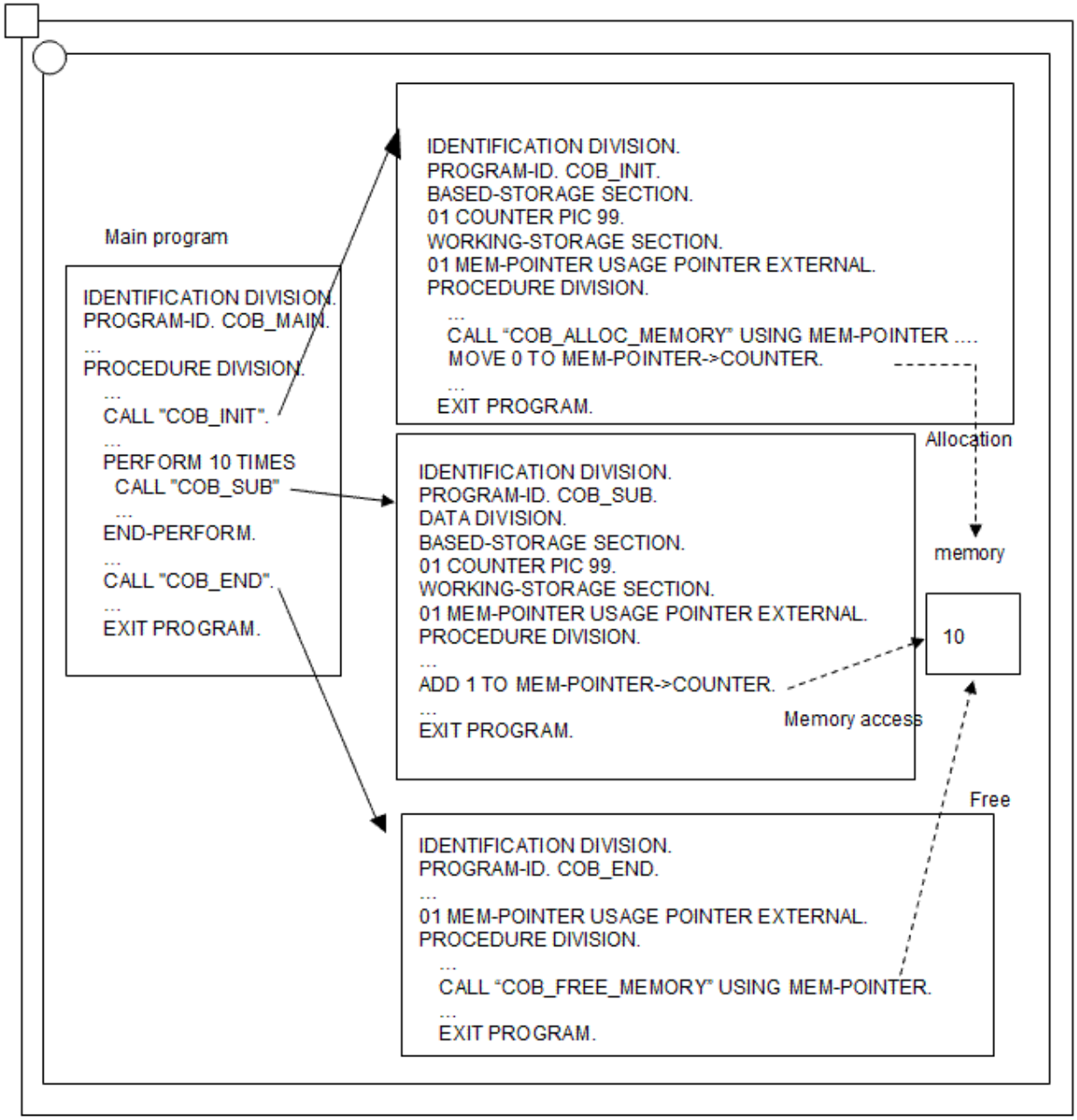
Program model	Type of memory specified with COB_ALLOC_MEMORY	Memory Freed
Process model	-	When execution environment is closed
Multithread model	Process	When execution environment is closed
	Thread	When run unit is ended

The type of memory specified with COB_ALLOC_MEMORY is ignored in the process model. In the multithread model, when memory is shared between threads, "process" is selected, and when memory used is for a specific thread, "thread" is selected. Refer to "16.2.2 Multithread and Process Model" for more information.

Process model

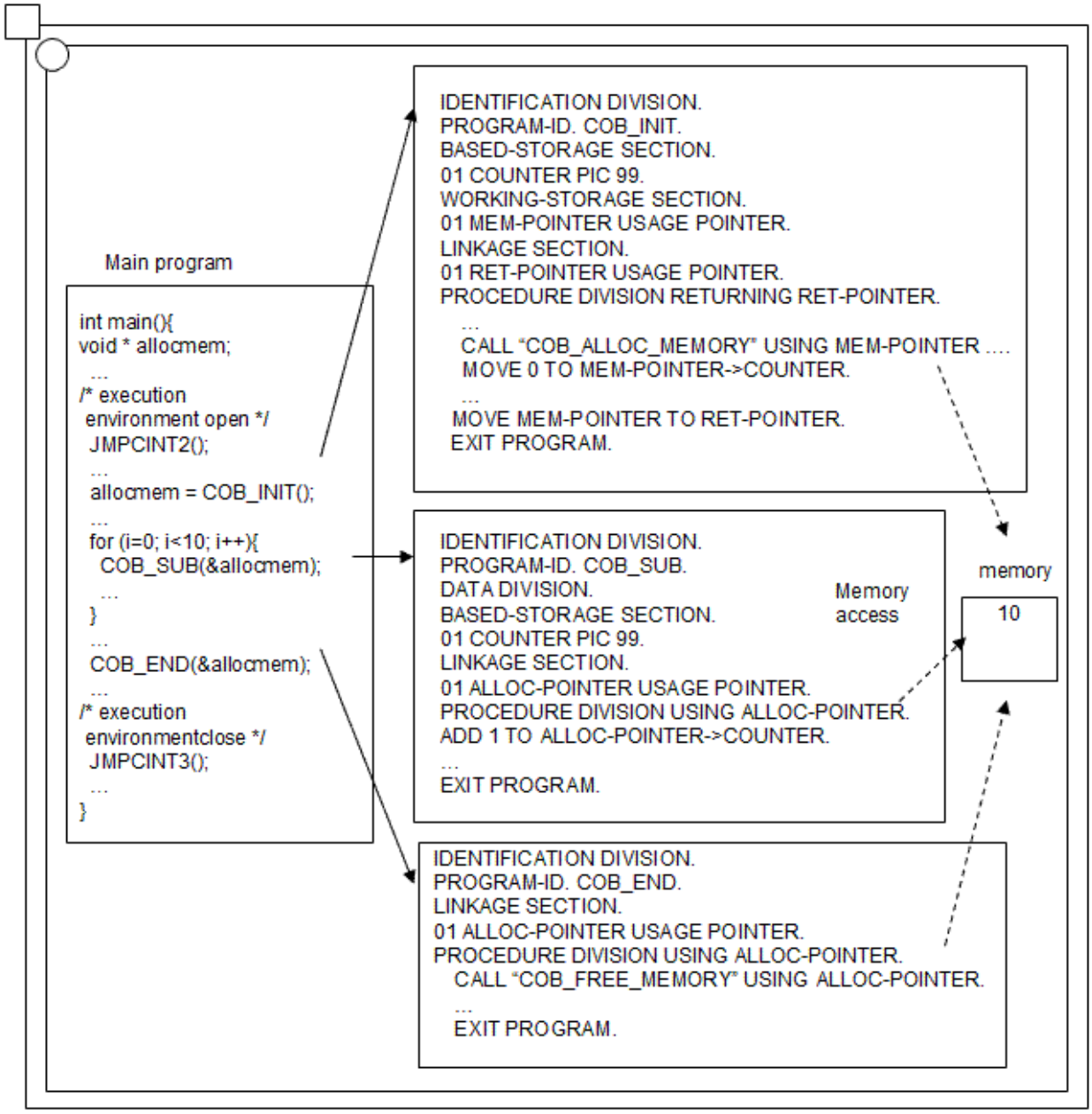
When the main program is COBOL

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 in the same run unit.



When the main program is not COBOL

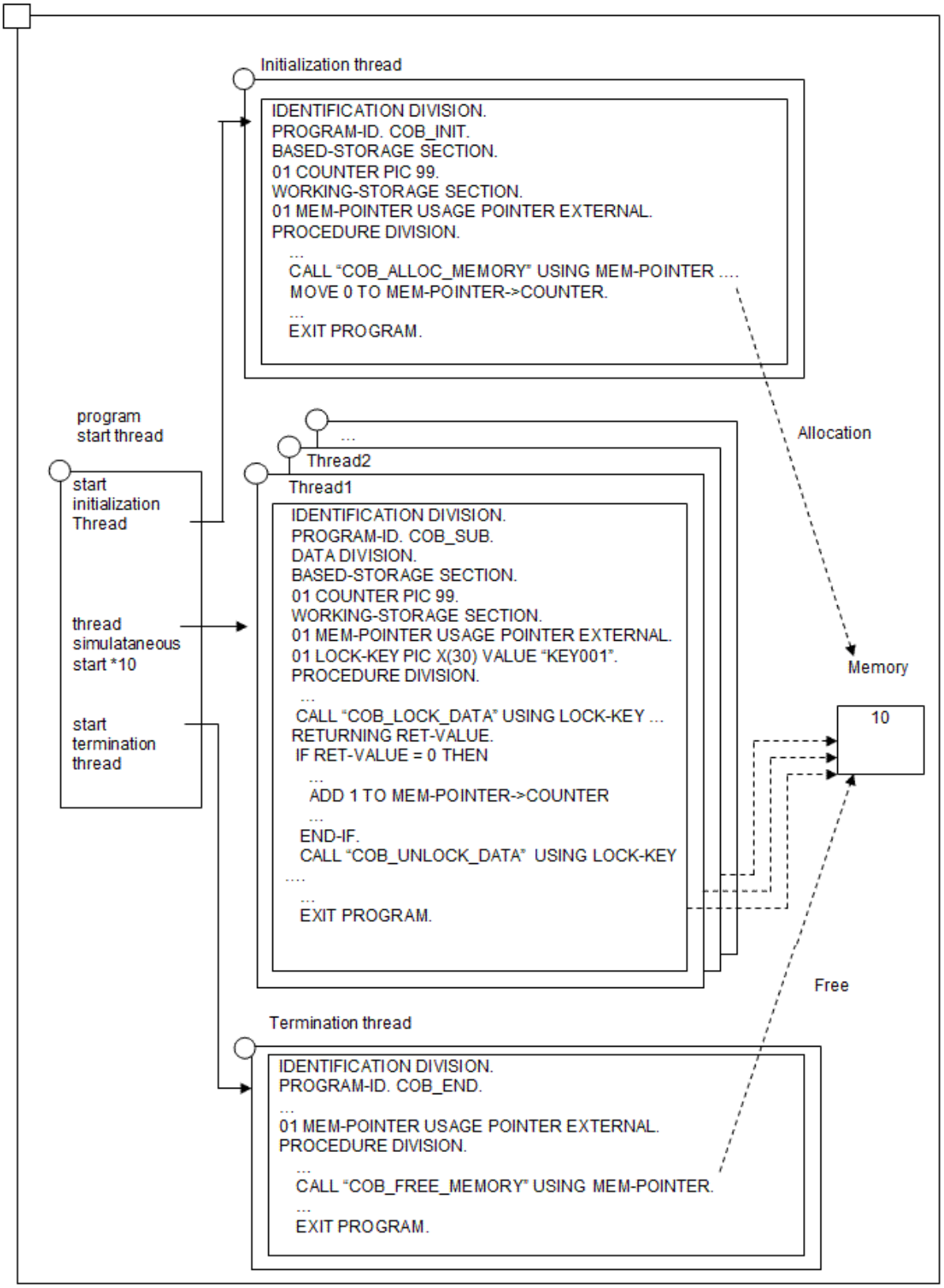
When the main program is a different language and operating two or more programs on the same run unit, operation similar to the COBOL program can be achieved by calling JMPCINT2 and JMPCINT3. When program COB_INIT is ended, the allocated memory is freed when neither JMPCINT2 nor JMPCINT3 are called, and the memory cannot be accessed from the program COB_SUB. Refer to "G.2 Subroutines Used in Linkage with another Language" for JMPCINT2 and JMPCINT3 usage.



Multithread model

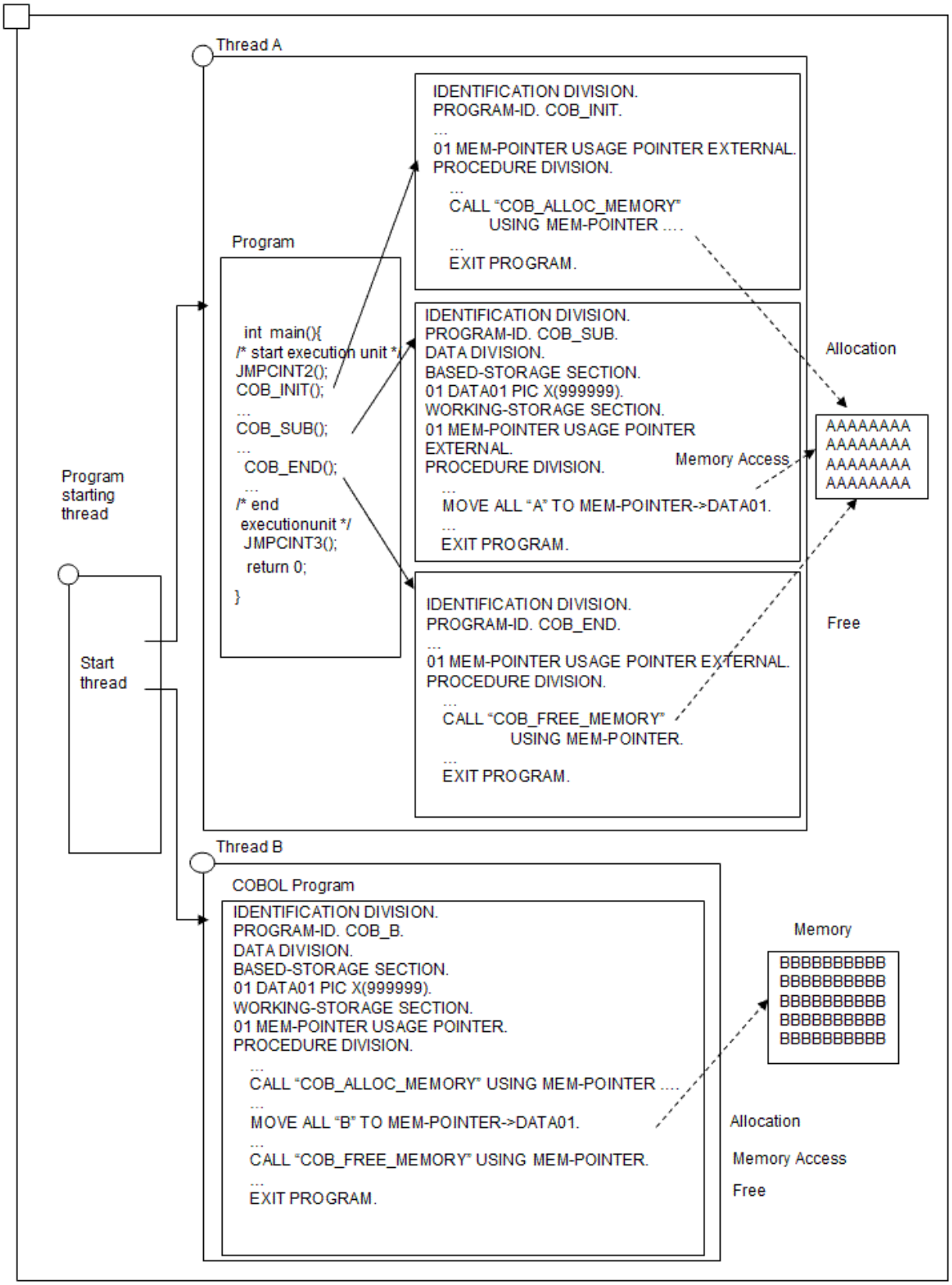
Process specification

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 the figure below, synchronous control by the data lock subroutine is needed. Refer to "16.4 Resource Sharing Between Threads" for details.



Thread specification

In thread A program COB_INIT and thread B program COB_B, COB_ALLOC_MEMORY is called 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 upon the call of JMPCINT3, when the main program is ended. Use "Thread" when memory used is for a specific thread.



Note

To call a subroutine using a dynamic program structure, you must use the following entry information file. For details about entry information file, refer to "4.1.3 Subprogram entry information".

```
[ENTRY]
Subroutine-name=librcobol.so
```

G.4.1 COB_ALLOC_MEMORY

You can use the COB_ALLOC_MEMORY subroutine 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. You can specify the following value:

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, then the return value is 0. If the operation fails, then the return value is as follows:

-1: Parameter error

-2: Insufficient memory

-3: The run unit is ended

G.4.2 COB_FREE_MEMORY

You can use the COB_FREE_MEMORY subroutine 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, then the return value is 0. If the operation fails, then the return value is -1. The reason is that the allocated memory is already freed, destroyed or not allocated using COB_ALLOC_MEMORY.

G.5 Subroutines to Forcibly end the Calling Process

COBOL offers the COB_EXIT_PROCESS subroutines to forcibly end the CALL process and all of its threads.

- The process is ended normally

  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 SIGABRT signal is issued, and the process is abnormally ended

  The abort function (SIGABRT signal) is issued and the process is abnormally ended. It can be used to analyze the core file.

Use subroutine COB_EXIT_PROCESS only when an acute problem is detected in the application. In the multi-thread environment, this routine terminates the process without waiting for other threads to end. Therefore, problems such as files not closed while being accessed concurrently by other threads may occur.

G.5.1 COB_EXIT_PROCESS

COBOL offers the COB_EXIT_PROCESS subroutines to forcibly end the calling process and all its threads.

Call Format

```
CALL  "COB_EXIT_PROCESS" USING BY VALUE data-name-1
                               BY VALUE data-name-2
                         RETURNING data-name-3.
```

Explanation of Parameter

Data-name-1

```
01 data-name-1   PIC 9(9) COMP-5.
```

Specify how to end the calling process. You can specify the following value:

0: Terminate the calling process normally and return the value specified in data-name-2 to the parent of the calling process.

1: Terminate the calling process abnormally by sending the signal SIGABRT unless the signal SIGABRT is caught and the signal handler does not return.

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.

Explanation of Return value

Data-name-3

```
01 data-name-3   PIC S9(9) COMP-5.
```

If the operation succeeds, then there is no return value. If the operation fails due to parameter error, then the return value is -1.

Appendix H Incompatible Syntax Between Standard and Object-Oriented

Some Object-Oriented COBOL features cannot be used in class definitions or method definitions. This appendix explains the unavailable class definitions features or separately coded methods (using the prototype method definition feature).

H.1 Features Not Allowed in Class Definitions

Table below 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: - Relationship between an ALL literal 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.
Report Writer	The REPORT SECTION cannot be used in the DATA DIVISION of FACTORY, OBJECT, and METHOD definitions.
CHARACTER TYPE clause	The CHARACTER TYPE clause cannot be specified in the DATA DIVISION of FACTORY, and OBJECT definitions. However, the CHARACTER TYPE clause can be specified in the DATA DIVISION of METHOD 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. 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.

Feature	Description
	All names declared in the DATA DIVISION of FACTORY or OBJECT definitions are treated as global names.
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.

H.2 Additional Features Not Allowed in Separate Method Definitions

Table below lists the features that cannot be used in the method definitions separated by the PROTOTYPE declare.

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 of the CLASS definitions.
SPECIAL-NAMES paragraphs	The SPECIAL-NAMES paragraph cannot be specified in the ENVIRONMENT DIVISION of separated method definitions.
WRITE statements with ADVANCING	WRITE statements with an ADVANCING clause can only be used if one of the following conditions is satisfied: - PRINTER 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 Database Link

This appendix explains the notes on linking an external database to a COBOL program.

I.1 Functional Outline

The product provides you with the function that outputs compilation error messages for a COBOL program including embedded SQL statements (COBOL program used as input of the precompiler) or debugs the COBOL program. This enables you to develop a program that links the database efficiently.

The databases that can be linked with COBOL are as follows:

- Oracle
- Symfoware

I.1.1 Oracle Link

For an Oracle link, a dedicated tool for linking with COBOL (insdbinf) is provided. The tool insdbinf embeds line number information in a COBOL program. Compiling the source program with the line number information embedded with COBOL enables the output of error messages for and debugging of a COBOL program including embedded SQL statements.

How to use the line-number-information-embed-tool

Name

The insdbinf tool inserts the line number information for COBOL in a file that Oracle Pro*COBOL outputs.

Syntax

```
insdbinf [-I include-path] ...                                          \
         [-S search-rule] -f source-file [RDB-generated-file] [-\?]
```

Option

-I include-path

Search path name of an include file.

Specify the same search path name as of the include file processed by Pro*COBOL. Specify all path names that are required to be searched.

If this is omitted, the current directory is searched.

Example

```
-I . -I ../inc -I /usr/include
```

-S search-rule

Search order of an include file.

Specify the extension of the include files processed by Pro*COBOL in the order they are searched.

If this is omitted, the character string "None" is searched.

Example

Searching in the order of (1) no extension and (2) .pco, .cob

```
-S //pco/cob/
```

Information

Use the name described in the include statement in the COBOL source program as an include file name.

-f source-file

Input file of Pro*COBOL.

Specify the input file of Pro*COBOL.

RDB-generated-file

Output file of Pro*COBOL.

Specify the output file of Pro*COBOL. If this is omitted, a file is read from the standard input.

-\?

Usage is displayed in a simplified form. No execution takes place.

I.1.2 Symfoware Link

If Symfoware/RDB is linked to COBOL, the options for linking to COBOL are directly supported in the Symfoware/RDB precompiler (Esql-COBOL). Unlike an Oracle link, tools (insdbinf) for linking to COBOL need not be used, For details on the Symfoware/RDB options for linking to COBOL, see the "Symfoware Server RDB User's Guide".

I.2 Flow to the Debugging of Embedded SQL Statements

You need to take an appropriate action for each database as shown in "I.1.1 Oracle Link" to enable the debugging of an embedded SQL statement. The following is the flow chart at the respective database link:

Figure I.1 Oracle link

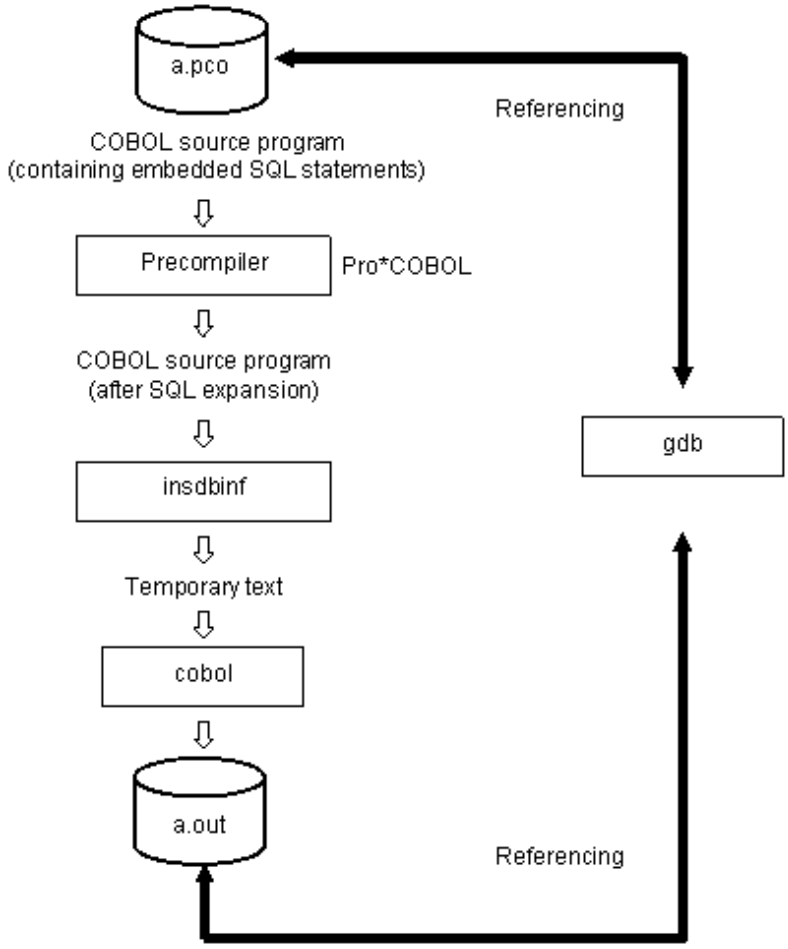
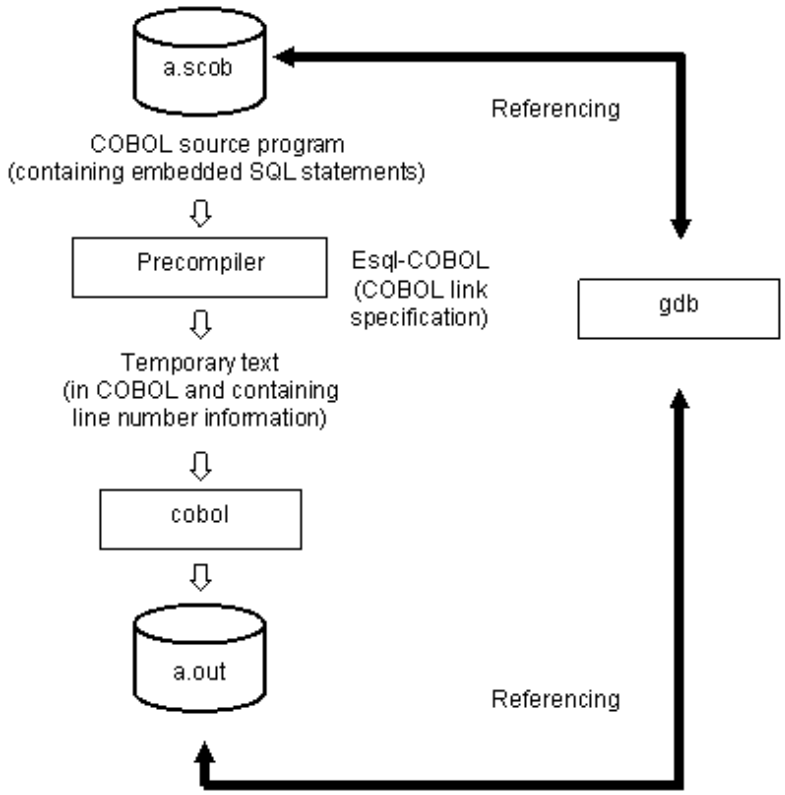


Figure I.2 Symfoware link



I.2.1 How to Debug Embedded SQL Statements

The remote debug function of NetCOBOL Studio supports the debugging function of a COBOL source program including embedded SQL statements. The following is the common debugging specification:

- The INCLUDE statement of an embedded SQL statement is handled in the same way as the COBOL COPY statement.

Precompilers (Pro*COBOL, Esql-COBOL) may not expand the SQL statements as a block of corresponding COBOL statements (WHENEVER, etc.). In this case remember that you cannot perform debugging operations including Set Breakpoint to those SQL statements. The SQL statements are handled the same way as comments.

I.3 Deadlock Exit

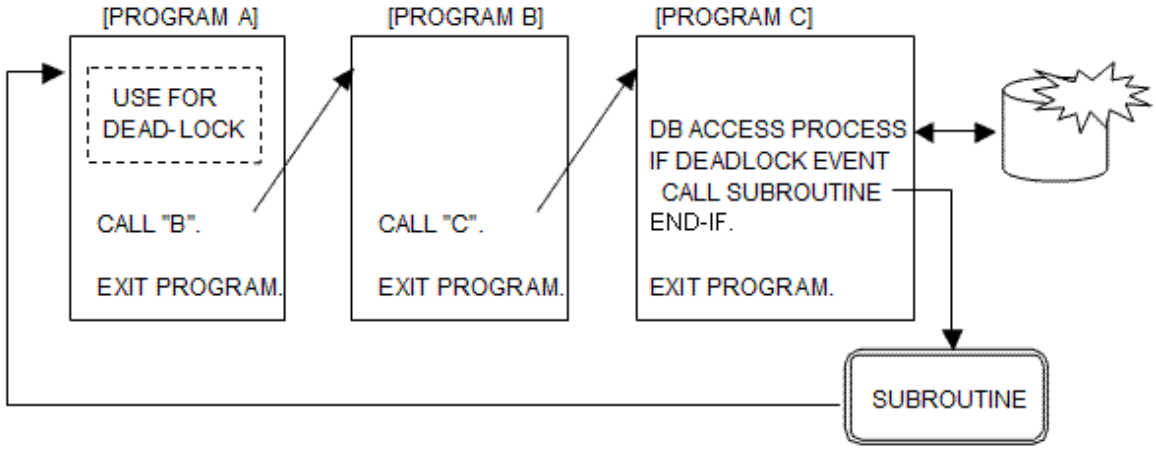
The deadlock state is the state in which an access conflict occurred as programs accessed a database and they are waiting for each other to release the occupied resource. If the deadlock state occurs, a deadlock event is posted from the database to the programs.

If the deadlock state occurs and a program has a coded deadlock exit, the deadlock exit schedule can be run. The processing procedure after a deadlock occurrence can be coded in the deadlock exit.

A deadlock exit is specified in the USE FOR DEAD-LOCK statement. For details on the USE FOR DEAD-LOCK statement, see the NetCOBOL Language Reference.

I.3.1 Overview of the deadlock exit schedule

The deadlock exit schedule returns control from the program where a deadlock event occurred to the program that has a coded deadlock exit.



The database is accessed with the precompiler. To run the deadlock exit schedule, the COBOL program notified of the deadlock event calls a subroutine provided by NetCOBOL.

The deadlock exit defined by a program is registered in the NetCOBOL runtime system when the program is executed. The registration of the deadlock exit defined by a program is cancelled when the EXIT PROGRAM statement of the program is executed.

I.3.2 Deadlock exit schedule subroutine

This subroutine returns control to the program that has a coded deadlock exit in the USE FOR DEAD-LOCK statement when a deadlock event has been posted to a program.

When the subroutine is called and the deadlock exit schedule is run, control returns to the deadlock exit coded in the program that called the subroutine or the nearest high-level program. Then, end processing equivalent to the EXIT PROGRAM statement in each program is executed between the program that called the subroutine and the program that has the coded deadlock exit to which control is returned.

Calling format

```
CALL  "COB_DEADLOCK_EXIT".
```

Parameter explanation

No parameter is required.

Return value

None

Calling condition

A return code is posted to SQLSTATE at the time of database access. If the return code specifies a value indicating a deadlock, the subroutine is called to return control to the program with a deadlock exit.

Note

- If neither the program that called the subroutine nor it's nearest high-level program specifies a deadlock exit, the deadlock exit schedule fails and an execution-time error (JMP0024I-U) is output to terminate processing abnormally.
- Any program written in another language that exists between the program that called the subroutine and the program that has the coded deadlock exit is not recovered. If processing is resumed after the deadlock exit, control is passed to the program written in another language. For this reason, the program written in another language must be re-entrant and have a structure that makes resource recovery unnecessary.
- If the subroutine is called from a program written in a language other than COBOL, operation is not guaranteed.
- The user has the responsibility of determining whether a deadlock event has occurred as described in "Calling condition."

- If the subroutine is called for a purpose other than taking action for a deadlock occurrence, operation is not guaranteed.
- The subroutine can be used in a multithread environment.

Notes on link

To create a program that uses the deadlock exit schedule subroutine, link librcobdlk.so as a linked shared library for an executable program or subprogram. librcobdlk.so is the shared library stored in the NetCOBOL runtime system installation directory.

 Example

Creating a shared library

```
cobol  -dy  -shared  -Tm  -o libDEAD1.so  -lrcobdlk  DEAD1.cob
```

I.3.3 Notes

- A procedure other than the declarative section must be executed with the GO TO statement after the deadlock process procedure is executed. If control is passed to the end of the deadlock process procedure, the program outputs the JMP0004I-U message and terminates abnormally.
- Database transactions are cancelled when a deadlock event is posted. The user has the responsibility for recovery of resources, such as the updating of files other than those cancelled database transactions.
- The GO TO statement and other such statements can be used to transfer control from inside the deadlock process procedure to a procedure other than the declarative section at the time of a deadlock exit. However, programs and the environment remain in the states that they were in immediately prior to control being passed to the deadlock process procedure. The user has the responsibility for restoring them to the appropriate states. For example, if the contents of a data item are changed or the branch destination of the GO TO statement is changed with the ALTER statement, they must be restored to appropriate values in the deadlock process procedure.
- The deadlock exit schedule subroutine must not be called from the USE section.
- If the LANGLVL(68/74) compile option is specified, the section or paragraph referenced with the PERFORM statement can be executed by another method (e.g., a GO TO statement). In such a case, do not write any statement that uses the database in that section or paragraph.

Appendix J National Language Code

This appendix explains how national language code is used in the product.

J.1 National Language Processing Code

J.1.1 Overview

This product supports Unicode and Shift-JIS as codes for the national language processing system. To specify the code system, set the value in the LANG system environment variable as indicated in the table below.

Table J.1 National Language Code Specification

Code	Value specified in LANG
Unicode	en_US.UTF-8
Shift-JIS	ja_JP.UTF-8
	zh_CN.UTF-8
	pt_BR.UTF-8

You do not have to be aware of character codes when creating, compiling, and executing a program in the system with only one code. However, when using systems with different codes, you should note.

The code in the product has the two concepts: the "code" in a program and "code" at runtime. Generally these two codes should match so that the national language processing takes place correctly.

The "code" in a program can be made a code different from the "code" at runtime by specifying the compilation option.

The code in a program, code at runtime, and matching of the codes are explained below.

Note

The value specified in the environment variable LANG applies only when performing national language processing. However, the product performs matching of the codes at execution of a program, so match the code in a program with the one at runtime except when the compilation option is specified.

J.1.2 Code in a Program and Code at Runtime

The code in a program is used to determine the code of the national data in national nonnumeric literals and national data items in the source program.

The code in a program is determined by the value specified in the environment variable LANG or the compilation option when compiling the source program with the cobol command.

The code at runtime is used to determine the code when the program displays or prints national data items. The code at runtime is determined by the value specified in the environment variable LANG when an executable program is started. Table below shows the specification of the codes in a program and at runtime:

Table J.2 Specification of the codes in a program and at runtime

Value specified in LANG (*1)	Encoding of data item	Codes at runtime	National language processing
en_US.UTF-8	Unicode	Unicode	Processed with Unicode.
ja_JP.UTF-8	Shift-JIS(*2)	Unicode	Program code is Shift-JIS.
zh_CN.UTF-8			Processed with Unicode to the system.
pt_BR.UTF-8			

Value specified in LANG (*1)	Encoding of data item	Codes at runtime	National language processing
Other than the above	ASCII		Not Supported.

*1 : The value specified in the environment variable LANG is enabled when:

- Code in a program: starting up a cobol command or;
- Code at runtime: starting up an executable file.

*2 : Specify the compiler option ENCODE(SJIS,SJIS).

Note

- When compiling a source program containing national language user-defined words or national nonnumeric literals, the code in the source program and the one at startup of a cobol command should match. The compilation result of the program is guaranteed only if they match.
- When invoking a COBOL program from C program using JMPCINT2 or JMPCINT3, the code at runtime is determined by the environment variable LANG when JMPCINT2 is invoked.
- There is no difference between internal encoding form when external excoding form is Unicode code set.

J.2 Notes on Transfer from Another System

J.2.1 Character codes of national and alphanumeric spaces

An incompatibility may occur in a comparison of spaces for any program transferred from the EBCDIC code system.

In the EBCDIC code system, a national character space is equivalent to two alphanumeric character spaces. This equivalency is not valid for other code systems.

The space character code depends on the code system as follows:

Code system	Alphanumeric space	National space	Main system
EBCDIC/JEF	X"40"	NX"4040"	OSIV system
EUC	X"20"	NX"A1A1"	UNIX system
Shift-JIS	X"20"	NX"8140"	Windows system
Unicode	X"20"	NX"3000"	Windows system, UNIX system

The following example shows a program in which an incompatibility occurs:

```
01  GR01.
  02  DATA1   PIC N(1).
01  DATA2     PIC N(1).
*>    :
PROCEDURE DIVISION.
*>    :
    MOVE SPACE TO GR01.  *> Moves the alphanumeric space.
    MOVE SPACE TO DATA2. *> Moves the national space.

    IF  DATA1 = DATA2 THEN
      DISPLAY "EQUAL"
    ELSE
      DISPLAY "NOT EQUAL".
```

When the above program is executed, "EQUAL" is displayed for the EBCDIC code system, and "NOT EQUAL" is displayed for the Shift-JIS and Unicode code systems.

Appendix K ld Command

This appendix explains the descriptions of the ld command syntax and how to use it when combining re-locatable programs generated by the COBOL compiler.

K.1 Command Syntax

```
$ ld [option-list] startup-routine file-name ...  library-name
```

Operand

option-list

See the manual of the ld command for the details of the options.

startup-routine

Specify the following files:

- /usr/lib64/crti.o
- /usr/lib64/crtl.o
- /usr/lib64/crtn.o

file-name

Specify all object file names to link for the static linkage and specify only the object file name of the main program for the dynamic linkage.

library-name

Specify one of the following libraries at the end of a command line with the -l option when creating an executable program.

-L /opt/FJSVcbl64/lib must be specified before the specified -l option.

Function	Library		Required
	At linkage of a process model program	At linkage of a multithread program	
Executable program with a dynamic link structure.	Shared object file of a subprogram (process model program).	Shared object file of a subprogram (multithread program).	
COBOL runtime	librcobol.so libFJBASE.so (*1)		Yes
Deadlock exit schedule subroutine	librcobdlk.so		
C standard library (system)	libc.so		
Thread library (system)	--------	libpthread.so	Yes
Dynamic link library (system).	libdl.so		Yes

*1: libFJBASE.so is required when an object-oriented program written in COBOL is linked.

Note

- When linking libpthread.so and libc.so in a multithread program, specify so that libpthread.so is linked first.

- Do not specify a library that does not support multithreads with the -l option when linking a multithread program.

K.2 How to Use the ld Command

This section explains how to create the executable program with the ld command from the re-locatable program created with the cobol command using examples.

K.2.1 When using the cobol command to link

As explained in "K.1 Command Syntax", when linking with the ld command, you need to specify library subroutines provided by COBOL in the ld command.

The following figures compare the linkage with the cobol command with the one with the ld command:

Using the cobol command to compile and link the program all at once.

```
$ cobol -dy -M -o P1 P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
```

Using the cobol command to compile the program, then link the program with the cobol command.

```
$ cobol -c -M -o P1 P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ cobol -dy -o P1 P1.o
```

Using the cobol command to compile the program, then link the program with the ld command.

```
$ cobol -c -M P1.cob
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2
  /usr/lib64/crti.o /usr/lib64/crt1.o /usr/lib64/crtn.o P1.o
  -Bdynamic -L/opt/FJSVcbl64/lib -lrcobol -ldl -lc -o P1
```

- cobol command
  - Input
    - P1.cob (source file)
  - Output
    - P1.o (object file)
  - Option
    - -M (Specifying the main program)
    - -c(Specifying to compile only)
- ld command
  - Input
    - crti.o crt1.o crtn.o (startup-routine)
    - P1.o(object-file)
    - llibrcobol.so libdl.so (library)
  - Output
    - P1 (executable-file)
  - Option
    - -dy (dynamic-linkage-specification)
    - -o(executable-program-output-destination)

- -L (library-search-path-name)
- -l (library-to-be-linked)

K.2.2 How to use the ld command by program structure

This section explains how to use the ld command when creating an executable program with a simple structure, dynamic link structure, or dynamic program structure.

Creating an executable program with a simple structure

The following is an example of the ld command when creating an executable program with a simple structure in the main program (P1) and subprogram (SUB):

```
$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -oP1
  /usr/lib64/crti.o /usr/lib64/crt1.o /usr/lib64/crtn.o P1.o SUB.o
  -Bdynamic -L/opt/FJSVcbl64/lib -lcobol -ldl -lc
```

Creating an executable program with a dynamic link structure

The following is an example of the ld command when creating an executable program with a dynamic link structure in the main program (P1) and subprogram (SUB):

[Creating the shared object file of a subprogram]

```
$ ld -shared -o libSUB.so SUB.o -L/opt/FJSVcbl64/lib -lrcobol
```

Execute the ld command to create the shared object program of a subprogram.

[Creating an executable file]

```
$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -oP1
  /usr/lib64/crti.o /usr/lib64/crt1.o /usr/lib64/crtn.o
  P1.o -L. -lSUB
  -Bdynamic -L/opt/FJSVcbl64/lib -lrcobol -ldl -lc
```

The ld command is executed, creating an executable program.

Creating an executable program with a dynamic program structure

The following is an example of the ld command when creating an executable program with a dynamic program structure in the main program (P1) and subprogram (SUB):

[Creating the shared object file of a subprogram]

```
$ ld -shared -o libSUB.so SUB.o -L/opt/FJSVcbl64/lib -lrcobol
```

Execute the ld command to create the shared object program of a subprogram.

[Creating an executable file]

```
$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -oP1
  /usr/lib64/crti.o /usr/lib64/crt1.o /usr/lib64/crtn.o
  P1.o -Bdynamic -L/opt/FJSVcbl64/lib -lrcobol -ldl -lc
```

The ld command is executed. The shared object program of the subprogram does not need to be linked to create an executable program having the dynamic program structure.

Appendix L Using the make Command

This appendix explains how to use the make command in the COBOL development.

L.1 About the make Command

Using the make command enables you to develop a program easily and reliably when handling more than one dependent resource in the COBOL development, especially when you use the object-oriented programming features. It is recommended that you consider using the make command to manage complex relationships between several resources.

If cobmkmf is used, Makefile is automatically created and processed using only simple resource configuration definitions. Therefore, the user need not know about Make to develop programs.

See the main manuals and documentation of the system for details of the make command and Makefile.

L.2 How to Write a Makefile

This section explains how to write a Makefile in COBOL.

L.2.1 Basic Syntax

Basically, in Makefile the following syntax is repeated:

```
target  : dependent-file ...
        command ...
        :
```

target:

Specifies the name of a file to be created.

dependent-file:

File that is required to create a target file. The creation of a target file takes place if there is no target file or the last update date of a file described here is newer than the one of the target file. In addition, if the dependent file described here is defined as a target file, the processing for the target file is prioritized.

command:

Specifies a command to create a target file.

 Example

Sample syntax of linkage rule in COBOL

```
executable-program-name :object-file-name ...
    cobol -o executable-program-name object-file-name ...
```

Sample syntax of compilation rule in COBOL

```
object-file-name : COBOL-source-file-name library-file-name ...
    cobol -c COBOL-source-file-name
```

L.2.2 Dependency of COBOL Resources

You need to define the dependency to the following resources in COBOL:

Target

Dependent file

Executable

Object, Shared object

Shared object

Object, Shared object

Object

Source, Library, Descriptors, Dependent repository, Option file

Target Repository

Source, Library, Descriptors, Dependent repository, Option file or object

Object with separated methods

Source, Library, Descriptors, Dependent repository, Option file, Object in a class with separated methods defined.

L.2.3 Dependency at when Classes are Cross-referenced

When classes are cross-referenced, the dependency is mutually related. Therefore, you cannot create the repository file required at compilation or the shared object required at linkage with a simple dependency. To avoid this, you need to compile to create a repository file before the compilation and to set up a link without any dependencies before the linkage.

Follow the steps below to share the class object creation with cross-reference:

1. Create repository files.

   There may not be repository files required for compilation. Compile classes with the compiler option "CREATE(REP)" to create repository files.

   When the parent class is referring to a child class, not a cross-reference, compile the parent class then the child class with the option "CREATE(REP)" to create repository files.

2. Create object files.

   Compile classes with the compiler option "CREATE(OBJ)" to create object files.

3. Create a shared object without a shared object linked.

   There may not be shared objects required for linkage. Create a shared object without the -l option at the linkage of an object.

4. Create a shared object with a shared object linked.

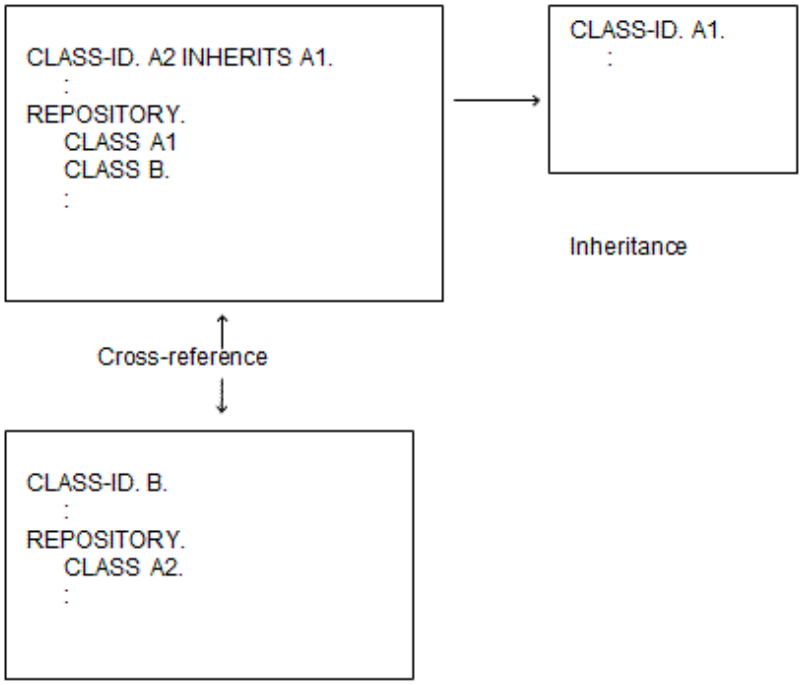
   Link an object with the -l option to create a shared object.

You need to use a tentative link identification file to distinguish steps 3 and 4 automatically with the Makefile.

Write the Makefile as follows when the classes are cross-referenced:

```
Repository file:  Source   Library   Various-descriptors  Inheritance-class-repository
        cobol -c -WC,"CREATE(REP)" Source

Object:  Source  Library  Various-descriptors  Dependent-repository   Option-file
        cobol -c -WC,"CREATE(OBJ)" Source

Temporary-link-identification:  Object
        cobol -shared -o Shared-object Object
        sleep 1
        touch Temporary-link-identification

Shared object:  Object  Dependent-temporary-link-identification
        cobol -shared -o Shared-object -l Shared-object Object
        touch Temporary-link-identification
```

For example, Class A2 that inherits Class A1 is cross-referenced with Class B, the Makefile is as follows:



Example of Makefile:

```
.SUFFIXES :
.SUFFIXES : .cob .rep .o .so

all: libA1.so libA2.so libB.so

### 1. Creating a repository file
.cob.rep:
         cobol -c -WC,"CREATE(REP)" $<

A2.rep: A1.rep
         cobol -c -WC,"CREATE(REP)" $<

### 2. Creating an object file
A1.o: A1.cob
         cobol -c -WC,"CREATE(OBJ)" A1.cob

A2.o: A2.cob A1.rep B.rep
         cobol -c -WC,"CREATE(OBJ)" A2.cob

B.o: B.cob A2.rep
         cobol -c -WC,"CREATE(OBJ)" B.cob

### 3. Creating a shared object to which no shared object is linked
A2.1: A2.o
         cobol -shared -o libA2.so A2.o
         sleep 1
         touch $@

B.1: B.o
         cobol -shared -o libB.so B.o
         sleep 1
         touch $@

### 4. Creating a shard object to which a shared object is linked
libA1.so: A1.o
         cobol -shared -o $@ A1.o
```

```
libA2.so: A2.o B.1 libA1.so
         cobol -shared -o $@ -L. -lA1 -lB A2.o
         touch A2.1

libB.so: B.o A2.1
         cobol -shared -o $@ -L. -lA2 B.o
         touch B.1
```

L.2.4 Makefile-Creation-Support Commands

The following commands are supported for creating a Makefile:

Command	Purpose for use
cobmkmf	Creating a Makefile for COBOL.
cobdepend	Investigating the dependency of COBOL source programs.
pmgr_rename	Converting file names.
pmgr_chsuffix	Converting extensions.

See the main manuals for details of the commands.

L.2.5 Sample Makefile

See the Makefile created with the cobmkmf command or a Makefile attached to each sample program for the syntax of a Makefile.

Appendix M COBOL coding techniques

This section describes techniques for coding COBOL programs.

M.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.

M.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.

M.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.

M.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.

M.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.

M.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.

M.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.

M.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.

M.2 Notes on numeric items

To avoid the occurrence of invalid data, review the following notes.

M.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).
      ...
         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).
      ...
         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
```

M.2.2 Large value binary items

Value range of binary item

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. See "A.2.51 TRUNC (Truncation Operation)"

Handling of number of digits for Binary item

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.

M.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.

M.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.
   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.

M.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.

Appendix N 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.

N.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 the resources by OS functions and programs. In particular, retain important resources in an intranet environment in which a firewall has been installed. The same is true if you are using the Web link function that offers COBOL on a Web server that has been installed outside an intranet environment. Make sure that you retain the important data resources in an 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 the files by OS functions.

N.2 Guidelines for Creating Applications

Consider the following before creating applications that involve security issues.

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.

Information

Cross-site scripting

Cross-site scripting refers to cases where input data containing JavaScript or other script code is embedded in HTML coding as output data without being checked by a program and the script code is executed by a client using that HTML coding for display. If malicious script code is input, cookie data is intercepted or modified, likely leading to authentication acceptance through cookies and/or session takeover. In addition to such script code, HTML tags can be used to display HTML pages different from those intended.

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 the log by using 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.

N.3 Remote Debug Function of NetCOBOL Studio

Although this product offers the remote debug function of NetCOBOL Studio for debugging programs that operate on other computers on the network, remote debug function of NetCOBOL Studio has not been designed or manufactured for use on the Internet. Use it in an environment that is not connected to the Internet, or in an intranet environment in which a firewall has been installed and which has been constructed to prevent security breaches.

Index

[S]

[T]

[U]

[V]