UNISYS

Product Information Announcement

o New Release ● Revision o Update o New Mail Code

Title

MCP/AS Binder Programming Reference Manual (8600 0304–301)

This announces a retitling and reissue of the *ClearPath HMP NX and A Series Binder Programming Reference Manual*. No new technical changes have been introduced since the HMP 1.0 and SSR 43.2 release in June 1996.

To order a Product Information Library CD-ROM or paper copies of this document

- United States customers, call Unisys Direct at 1-800-448-1424.
- Customers outside the United States, contact your Unisys sales office.
- Unisys personnel, order through the electronic Book Store at http://iwww.bookstore.unisys.com.

Comments about documentation can be sent through e-mail to **doc@unisys.com**.

Announcement only:	Announcement and attachments:		
	AS200	System:	MCP/AS
		Release:	HMP 4.0 and SSR 45.1
		Date:	June 1998
		Part number:	8600 0304–301

MCP/AS

UNISYS

Binder

Programming Reference Manual

Copyright © 1998 Unisys Corporation.
All rights reserved.
Unisys is a registered trademark of Unisys Corporation.

HMP 4.0 and SSR 45.1 June 1998

Printed in USA
Priced Item 8600 0304–301

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of a duly executed agreement to purchase or lease equipment or to license software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED – Use, reproduction, or disclosure is restricted by DFARS 252.227–7013 and 252.211–7015/FAR 52.227–14 & 52.227-19 for commercial computer software.

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the Business Reply Mail form at the back of this document or by addressing remarks to Software Product Information, Unisys Corporation, 25725 Jeronimo Road, Mission Viejo, CA 92691–2792 U.S.A.

Comments about documentation can also be sent through e-mail to **doc@unisys.com**.

Unisys and ClearPath are registered trademarks of Unisys Corporation.
IInfoExec is a trademark of Unisys Corporation.
All other terms mentioned in this document that are known to be trademarks or service marks have been appropriately capitalized. Unisys Corporation cannot attest to the accuracy of this information. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Contents

Section 4. Binding Programs Written in the Same Language

Section 5. Binding Programs Written in Different Languages

Contents

Figures

Figures

Tables

About This Manual

Purpose

This manual explains how to use the Binder compiler to insert a module from a separately compiled program into another separately compiled program.

Scope

This manual begins with an introduction to the process of binding. The main text includes information, syntax, and examples for binding programs and libraries written in the same language and in a variety of different languages.

Audience

Programmers of all experience levels can use this manual.

Prerequisites

You must be familiar with the languages in which the programs you are binding are written.

How to Use This Manual

Read the first section of this manual to understand the binding function and process. You can use the rest of the manual as a reference tool to obtain more information for your specific program binding needs.

The syntax of Binder statements is presented in this manual in *railroad* syntax diagram form. If you are unfamiliar with this notation, see Appendix C for a complete explanation.

Organization

This manual consists of eight sections, three appendixes, and an index. The content of the sections and appendixes is described as follows:

Section 1. Understanding the Binding Process

This section explains the overall binding process.

Section 2. Binder Language Constructs

This section describes the elements that form the most primitive structures of the Binder language.

Section 3. Binder Statements

This section provides the syntax and function of the language elements used with Binder.

Section 4. Binding Programs Written in the Same Language

This section describes the procedures and techniques required to perform *intralanguage binding*, which is the process of binding programs written in the same language.

Section 5. Binding Programs Written in Different Languages

This section describes the procedures and techniques required to perform *interlanguage binding*, which is the process of binding programs written in different languages.

Section 6. Binding Intrinsics

This section describes the binding procedures that are required to create and bind intrinsic files.

Section 7. Binding Programs That Access Databases

This section explains how to bind programs that access SIM or DMSII databases.

Section 8. Printing Binding Information

This section describes how to use the PRINTBINDINFO utility to print an analysis of the binding information of a code file.

Appendix A. Warning and Error Messages

This appendix lists the various warning and error messages and their meanings, and provides solutions for the errors when applicable.

Appendix B. Using Binder Control Record Options

This appendix describes how to use Binder control record options to control the processing of Binder input files and the content of the resulting bound code file.

Appendix C. Understanding Railroad Diagrams

This appendix describes the notation used throughout this manual to represent the syntax of the Binder language.

Related Product Information

Unless otherwise stated, all documents referred to in this publication are MCP/AS documents. The titles have been shortened for increased usability and ease of reading.

The following documents are included with the software release documentation and provide general reference information:

- The *Glossary* includes definitions of terms used in this document.
- The *Documentation Road Map* is a pictorial representation of the Product Information (PI) library. You follow paths through the road map based on tasks you want to perform. The paths lead to the documents you need for those tasks. The Road Map is available on the PI Library CD-ROM. If you know what you want to do, but don't know where to find the information, start with the Documentation Road Map.
- The *Information Availability List* (IAL) lists all user documents, online help, and HTML files in the library. The list is sorted by title and by part number.

The following documents provide information that is directly related to the primary subject of this publication.

CANDE Operations Reference Manual

This manual describes how CANDE operates to allow generalized file preparation and updating in an interactive, terminal-oriented environment. This manual is written for a wide range of computer users who work with text and program files.

Work Flow Language (WFL) Programming Reference Manual

This manual presents the complete syntax and semantics of WFL. WFL is used to construct jobs that compile or run programs written in other languages and that perform library maintenance such as copying files. This manual is written for individuals who have some experience with programming in a block-structured language such as ALGOL and who know how to create and edit files using CANDE or the Editor.

Section 1
Understanding the Binding Process

What Is Binder?

Binder is a utility that lets you permanently insert a module from one compiled program into another compiled program. The module you want to insert is called a *subprogram*. The program in which you are inserting the subprogram is called the *host program*. Binder lets you combine subprograms and host programs written in the same language or in a variety of different languages. Table 1–1 shows the allowable binding combinations.

By using Binder, you can change or correct an existing program without having to rewrite or recompile the entire program. For example, if a program accesses several subprograms, and some require changes, you can revise and recompile only the subprograms that need changes, and then use Binder to combine the subprograms into one resultant program. This process saves computer time in recompiling and programmer time in rewriting.

Binder also allows you to use a standard set of subprograms with multiple other programs. You need to write the subprograms only once. Then, you can bind them into the other programs whenever you need to do so.

Binder Code File Restrictions

You cannot bind code files that are more than three system software releases older than the release level of the Binder program with which you are working. For example with the Mark 3.9 release of Binder, you can only bind code files of Mark 3.6 or later. If you use a code file that is too old, Binder flags the file with an error message and terminates.

If you use your compiler to generate code that runs on a restricted set of computers, the resulting bound code file will run only on the computers on which the host program and the bound subprograms run. For example, if one code file runs on an A 4 and an A 16 and another code file runs only on an A 4, the bound code file will run only on the A 4.

Table 1–1. Allowable Binding Combinations

Subprogram Language	Host Program Language							
	ALGOL†	C	COBOL	FORTRAN	FORTRAN77	NEWP‡	Pascal□	PL/I
ALGOL†	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
C		Yes						
COBOL	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
FORTRAN	Yes	Yes	Yes	Yes	Yes	Yes		
FORTRAN77	Yes	Yes	Yes	Yes	Yes	Yes		
PL/1								Yes

† All references to ALGOL include the various extensions of ALGOL, such as BDMSALGOL, DCALGOL, and DMALGOL.

‡ The NEWP Master Control Program (MCP) can serve only as a host program in binding.

□ Pascal programs can serve only as host programs in binding.

Binder Input Files

In a normal execution, you supply Binder with the following input:

- A primary input file (optional)
- A compiled host program
- One or more externally compiled subprograms

The Primary Input File

The primary input file is an optional file that consists of Binder statements and Binder control records. You can use Binder statements to indicate the titles of the subprograms and the title of the host program to be bound. You can also use Binder statements to exclude certain subprograms from the binding process. You can use Binder control records to control the way Binder processes the subprogram and the host program, and to determine the content of various files produced during binding. Binder statements are described in Section 3. Binder control records are described in Appendix B.

The internal name of the primary input file is CARD. If you initiate the bind from WFL, the file kind is READER. If you initiate the bind from CANDE, the file kind is DISK, unless you use a file equation.

The Host Program

The host program is the code file to which subprograms can be bound. A host program must contain the first executable code segment of a program. A host program can be the resultant code file of a previous bind.

You can specify the title of the host file to Binder by using the Binder HOST statement (see Section 3) or by file equating file HOST in the WFL or CANDE syntax used to start Binder. The internal name of the host program is HOST. The file kind is DISK, unless you change the file kind with a file equation. You must always supply a host program, except when binding intrinsics. For details about binding intrinsics, see Section 6.

Some examples of host programs are as follows:

- An ALGOL outer block
- A FORTRAN program containing a main program
- The MCP
- A PL/I procedure
- A COBOL program compiled with the LEVEL option set to 2
- A previously bound program
- A FORTRAN77 program with $ BINDINFO set
- A Pascal program with modules declared EXTERNAL
- A C program containing the function, “main”

The Subprogram

A subprogram is a separately compiled program unit that exists externally to the host program. You must compile external subprograms with the appropriate language compiler before binding them to a host program. Note that a subprogram *cannot* be the resultant code file of a previous bind. Multiple subprograms can exist in a subprogram file.

External subprograms are referenced in the host program but have not yet been bound. You do not have to specify external subprograms in the BIND statement, because they are bound automatically by default.

Binding makes the subprogram a part of the host program. When you bind a new version of the subprogram, the new version replaces the existing version in the host program. This procedure is known as *replacement binding*.

Some examples of subprogram files are as follows:

- ALGOL procedures
- FORTRAN77 subroutines or functions
- Separately compiled procedures of the MCP
- Intrinsics
- PL/I procedures
- COBOL programs compiled with the LEVEL option set to a value greater than 2
- A compiled C module that does not define the function “main”

The ALGOL, FORTRAN, FORTRAN77, and PL/I compilers title subprograms compiled through WFL by replacing the identifier following the last slash of the code file title in the

WFL COMPILE statement with the subprogram name. In COBOL, the subprogram name is taken from the identifier following the last slash of the code file title in which the subprogram resides.

In ALGOL, FORTRAN, and FORTRAN77, a LIBRARY compiler control option is available that causes the compiler to place all subprograms compiled in a single compilation into one code file. The title of the code remains as specified in the COMPILE statement. The LIBRARY option is automatically set to TRUE when the compilation of an ALGOL program with one or more independent procedures is initiated by CANDE, or when the compilation of a FORTRAN or FORTRAN77 program with the SEPARATE compiler control option set to TRUE is initiated by CANDE.

Binder Output Files

Binder can produce three files during normal execution:

- A bound code file

  This is a file consisting of the host program and the subprograms bound into the host program.

  All of the deimplementation warnings produced for the individual code files are included in the bound code file. In addition, the bound code file might also contain unresolved references to external programs. Unresolved external references occur when subprograms referenced in the host do not get bound. Unresolved external references are discussed later in this section.

  The internal file name for the bound code file is CODE. The KIND attribute for the file is DISK; you cannot change the KIND attribute with a file equation.

  If the host and all subprograms have the same FILEKIND attribute (such as ALGOLCODE), that FILEKIND attribute is retained for the new bound codefile. Otherwise, the FILEKIND attribute is BOUNDCODE.

- An optional printer listing

  The contents of the printer listing vary depending upon the Binder control record options you specify. To produce a printer listing, include the LIST or TIME Binder control record option in the primary input file.

  The internal file name for the printer file is LINE. The file kind is PRINTER unless you change the file kind with a file equation.

- An optional error file

  The error file, labeled ERRORS by default, contains all the error messages produced during the binding process. To generate an error file, include the ERRORLIST Binder control record option in the input file you use to invoke Binder.

  If you initiate Binder from WFL, the file kind is PRINTER. If you initiate Binder from CANDE, the file kind is REMOTE, unless you change the kind with a file equation.

Avoiding Unresolved External References in the Bound Code File

A reference to an external subprogram is in a resolved state when the subprogram is successfully bound to the host program. An external reference is in an unresolved state when the subprogram does not get bound to the host program.

External subprograms do not get bound to the host program if

- Binder cannot locate the subprogram
- You use the Binder EXTERNAL statement in the host program to prevent the subprogram from being bound

Unresolved references to external subprograms are fatal to program execution if the program tries to access the unbound subprogram. Program execution is not affected if the program does not attempt to access the unbound subprogram.

You can help prevent fatal program errors due to unresolved external references by including the WAIT, STRICT, and LIST Binder control record options in the primary input file.

WAIT	Causes Binder to suspend binding when it cannot find a specified subprogram. You can then make the subprogram available and resume binding, or you can terminate Binder.
STRICT	Prevents the resultant code file from being locked if a specified subprogram is not bound.
LIST	Produces a printer listing that you can use to verify that all necessary subprograms have been bound before you attempt to execute the program.

Invoking Binder

There are two ways to invoke Binder:

- By using the CANDE command, BIND, to activate the primary input file (a work file or disk file containing Binder statements)
- By using a WFL job that contains Binder statements

Refer to Section 3 for the syntax and explanation of the Binder statements.

When the bind is complete, Binder gives the time of the compilation, as well as the compiler name and version number for the subprograms and the host program.

Invoking Binder from CANDE

Your primary input file must contain BIND statements to indicate the location of the subprograms to be bound. The input file can optionally indicate the name of the host

program to which the subprograms are being bound. If the input file does not contain the name of the host program, you must indicate the host program name by using a file equation in the CANDE BIND command.

For example, assume that you have the following CANDE file, named BOUND/LIB, as the primary input file:

```
HOST IS OBJECT/BOUND/LIB/HOST;
BIND SUBA FROM OBJECT/BOUND/LIB/PASSR;
BIND SUBB FROM OBJECT/BOUND/LIB/PASSR;
```

To invoke Binder, you would enter

```
BIND BOUND/LIB
```

Assume that the HOST statement was not included in the BOUND/LIB file, and instead, the file looked like the following:

```
BIND SUBA FROM OBJECT/BOUND/LIB/PASSR;
BIND SUBB FROM OBJECT/BOUND/LIB/PASSR;
```

In this case, the command to invoke Binder would be

```
BIND BOUND/LIB; BINDER FILE HOST=OBJECT/BOUND/LIB/HOST
```

For both of the preceding command examples, the resultant bound code file would be titled *OBJECT/BOUND/LIB*.

Refer to the *CANDE Operations Reference Manual* for the complete syntax and description of the BIND command.

Invoking Binder from WFL

You can list Binder statements in a WFL job, and then use the WFL job to initiate the bind, as shown in the following example.

```
? BEGIN JOB BIND/SYSTEM/MYLIB
  BIND SYSTEM/MYLIB BINDER LIBRARY;
  BINDER DATA
  HOST IS OBJECT/BOUND/LIB/HOST;
  BIND SUBA FROM OBJECT/BOUND/LIB/PASSR;
  BIND SUBB FROM OBJECT/BOUND/LIB/PASSR;
? END JOB.
```

The resultant bound code file would be titled *SYSTEM/MYLIB*.

Reserved Words

The following list contains words that are reserved for use in Binder syntax. You cannot use these words for any purpose other than that described in this manual.

BIND	FROM	OF
DONTBIND	HOST	PURGE
EXTERNAL	INITIAL	STOP
FOR	IS	USE

Binder Execution

Binder begins execution by reading the primary input file (CARD), if one exists. If Binder finds a primary input file, it processes and stores the Binder statements for future reference. If Binder detects any syntax errors during the processing, it terminates after reading the last input record of the file. If a primary input file does not exist, Binder attempts to open the host program and read the first record.

If the host program is not present or cannot be made present, the operating system discontinues Binder. If the host program is not a code file or is otherwise not suitable for binding, the appropriate error message appears and Binder terminates.

If Binder finds a host program, it locates and reads the Binder information contained therein. Binder determines if each named subprogram is bound or unbound, and then determines whether a statement from the primary input file applies to the subprogram. As a result of this examination, Binder takes the actions shown in Table 1–2.

Table 1–2. Binder Action on Subprograms Named in the Host Program

Primary Input File Statement	Bound Subprogram	External (Unbound) Subprogram
No statement	Ignores subprogram	Attempts to bind subprogram
DONTBIND statement	Ignores subprogram	Ignores subprogram
BIND statement	Binds subprogram and discards previously bound subprogram (*replacement binding*)	Binds subprogram

Binding Subprograms

When directed to bind a subprogram, Binder attempts to find the correct file where the subprogram resides. If the correct file is not present on disk and neither the STRICT nor WAIT options are specified, Binder ignores the subprogram, sends a message indicating that it was unable to access the file, and continues to look for other subprograms to bind. (For more information about the STRICT and WAIT options, refer to Appendix B.)

If Binder is directed to a host program or to the resultant code file of a previous bind, Binder sends a message indicating that the file is suitable only as a host and does not bind the given subprogram. Binder continues to look for other subprograms to bind.

When Binder finds a file containing a subprogram, it first verifies that the file contains the necessary information for binding. Binder also verifies that the subprogram matches the description of what is expected by the host. If the type of subprogram, its number or type of parameters, or its execution level does not match its declaration in the host, Binder discontinues binding the subprogram, returns to its previous level of binding, and continues to look for other subprograms to bind.

Encountering Errors

Once Binder finds a subprogram that matches the host description, any subsequent error conditions arising during the bind of that subprogram are usually fatal, and binding is discontinued.

Errors occur during the binding process when Binder finds a mismatch between the description of a global reference made in the subprogram and its corresponding description in the host program. Certain languages allow minor discrepancies in type matching, such as referencing a variable as a real number in the subprogram when it is declared as an integer in the host. However, more serious mismatches, such as referencing a single-precision variable as an array or calling another subprogram with the wrong number of parameters, are flagged as fatal errors.

Using Binder Efficiently

Most of Binder's execution time is used to perform input and output operations. For this reason, the most efficient way to use Binder is to maintain a host program that contains a completely bound program. When you need to update an existing subprogram, you change the code, recompile the subprogram, and then replace the existing subprogram in the host program by binding in the new version. This method, called replacement binding, requires that only two files be accessed, greatly reducing the I/O time used.

If your host program is not a completely bound program, you can waste a great deal of I/O time. For example, if you have an unbound host program and 250 files that contain subprograms to be bound to it, you have to rebind all the files each time you update one subprogram. To bind the host program and the subprograms together requires the opening and closing of 251 files with corresponding buffer allocations and deallocations, as well as the I/O time to read and write the files. You could reduce the number of files by using the LIBRARY option available in some compilers to combine several subprograms into one library code file. However, using this option is not as effective as maintaining a bound program.

Object-Code Efficiency

In general, the bound code file produced by Binder is equivalent to the code file produced by a language compiler. In the case of replacement binding, a process in which an existing subprogram is replaced by binding in a new subprogram, Binder reuses some segment

dictionary locations used exclusively by the subprograms being exchanged. However, code segments not needed by the new subprogram might remain in the bound code file. These obsolete code segments do not affect the execution of the bound code but do occupy storage space.

Once added to a bound program, items at lexical (lex) level 2 are never removed. For example, if a subprogram containing variables declared as OWN or STATIC is replaced, the lex level-2 locations for the one or more variables declared as OWN in the replaced subprogram are not reused. New lex level-2 locations can be allocated for the subprogram that replaces the existing subprogram. (Although variables declared as OWN exist at lex level 2, they are accessible only to the program unit that declares them; thus, if this program unit is replaced, the lex level-2 locations are inaccessible.)

Usually the unreferenced lex level-2 stack locations belonging to replaced subprograms cause very little overhead in execution or in core usage. However, if an initialized array, which is an array containing initial values other than 0 (zero), is declared as OWN, the code to initialize the array causes the array to be made present. Therefore, repeated replacement binding of a subprogram that contains initialized arrays can cause some additional core usage and can increase execution time.

Section 2
Binder Language Constructs

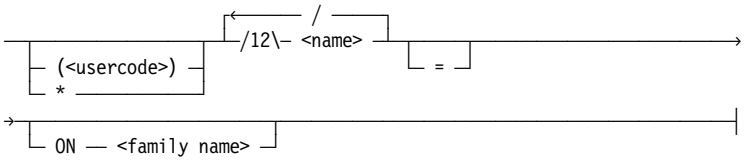
This section describes the syntactic items that appear in the syntax diagrams in Section 3 of this manual.

File Specifier

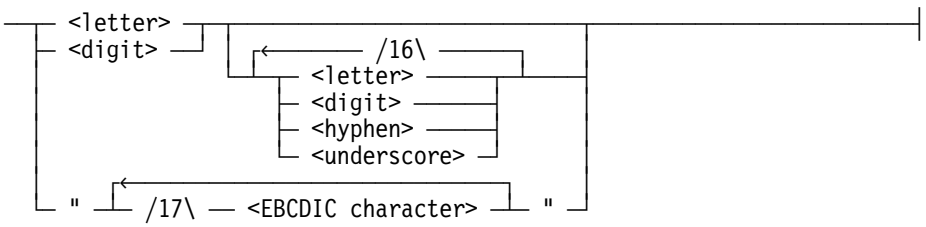
Use the file specifier construct to indicate the name of a file.

Syntax

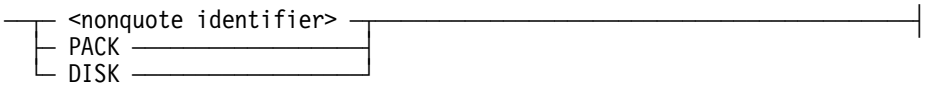
<file specifier>



<name>



<family name>



Explanation

<letter>	Any one of the 26 uppercase characters, A through Z
<digit>	Any character in the range 0 (zero) through 9
<hyphen>	The hyphen character (-)
<underscore>	The underscore character (_)

<name>	A string of characters used to identify an entity such as a file, a usercode, or a device group.
<EBCDIC>	Any EBCDIC character for which the hexadecimal code is greater than or equal to hexadecimal 40 and is not the EBCDIC quotation mark character (")
<usercode>	A name whose purpose is to establish user identity, to control security, and to provide for segregation of files
<family name>	An identifier that specifies the group of disk storage devices that function as one logical unit.
<nonquote identifier>	A string of 1 to 17 alphanumeric characters
alphanumeric character	Any of the characters A through Z or 0 (zero) through 9

Details

In a file specifier, all of the names except the last indicate the directory in which a file is located. The last name is the actual file name. For example, in the file specifier A/B/C, A/B is the directory name, and C is the file name.

The file specifier can be optionally preceded by a usercode (enclosed in parentheses) or by an asterisk (*). A family other than the default family (which usually is DISK) can be specified by using the suffix, ON <family name>.

The name and the family name can consist of 1 through 17 alphanumeric characters and cannot be split across input record boundaries.

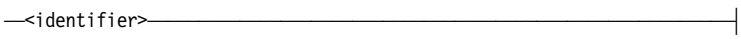
If you use an equal sign (=) as part of the directory name, Binder replaces the equal sign with the subprogram name. For example, if the directory name is *A/B/*= and the subprogram is *S*, Binder looks for a file titled *A/B/S*. If multiple files have the same directory name, as in A/B/SUB, A/B/PROG, A/B/ALG, Binder examines each of the specified files in order of appearance to determine if the file contains the subprogram to be bound.

Examples

```
A/B/=

(MYUSERCODE)TEST/=

*= ON TESTPACK

A/B/C

FILEID1/FILEID2/FILEID3 ON MYPACK
```

Identifier

Use the identifier construct to indicate the name of a file, subprogram, or program variable.



Explanation

An identifier consists of any combination of the following characters, optionally enclosed in quotation marks. You cannot split an identifier across input record boundaries.

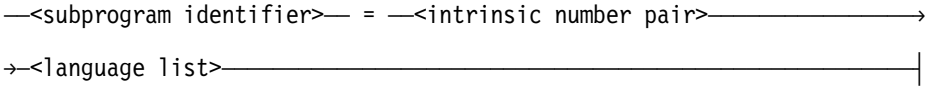
A through Z	- (hyphen)
a through z	@ (commercial at)
0 (zero) through 9	/ (slash)
_ (underscore)	$ (dollar sign)
’ (single quote)	# (pound sign)
. (period)	

Intrinsic Specification

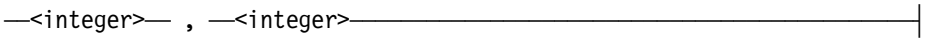
Use the intrinsic specification construct to indicate the name of an installation intrinsic to be bound.

Syntax

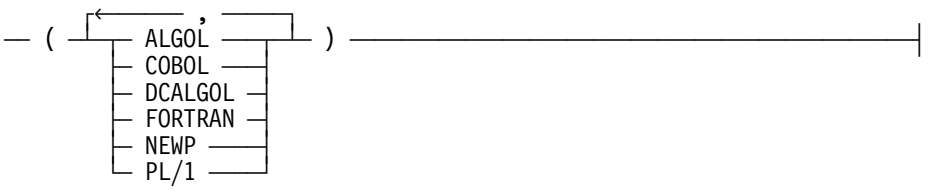
<intrinsic specification>



<intrinsic number pair>



<language list>



Explanation

<intrinsic number pair>	Specifies an intrinsic number pair. The first integer of the intrinsic number pair specifies an installation number, which can range in value from 0 through 2046; however, numbers 0 through 99 are reserved for system use. The second integer specifies an intrinsic number, which can range in value from 0 through 8191. No two intrinsics within an intrinsic file can have the same intrinsic number pair.
<language list>	Specifies a list of those compilers authorized to reference a given intrinsic. A referencing language is not necessarily the same as the language in which the intrinsic is written. The DCALGOL language identifier allows a specified intrinsic to be accessed by the DMALGOL compiler as well as by the DCALGOL compiler.

Details

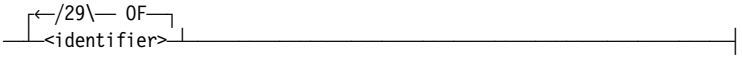
Standard system intrinsics that are referenced as EXTERNAL in a program are automatically bound. Thus, you do not need to declare such system intrinsics in a BIND statement. See Section 6 for details on binding intrinsics.

Examples

```
$ SET INTRINSICS
 BIND = FROM INTR/=;
 BIND MYSIN = 101, 1 (ALGOL,FORTRAN) FROM INTL/=;
 BIND COFFEE = 102, 2 (COBOL) FROM POT;
 STOP;
```

Subprogram Identifier

Use the subprogram identifier construct to indicate the name of a subprogram.



Explanation

The identifier construct is defined earlier in this section.

Details

If you have subprograms with the same name, you must use identifiers to uniquely identify, or qualify, the subprograms. A structure block type or a connection block type can be used as an identifier to uniquely qualify a subprogram.

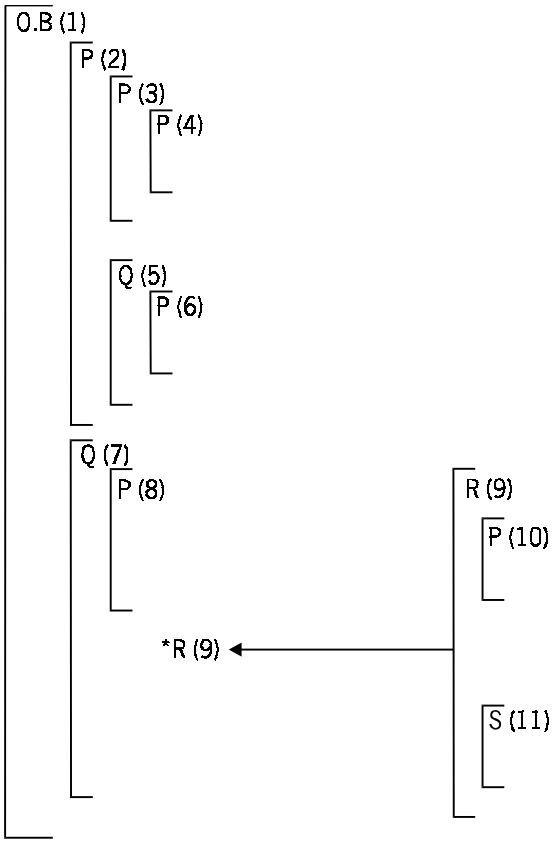
A subprogram identifier can contain a maximum of 30 identifiers. A level of nesting cannot be skipped when a subprogram is qualified.

When a subprogram identifier is used in a Binder statement, the Binder statement is applicable to all subprograms that fit the qualifications of the subprogram identifier.

Figure 2–1 illustrates the nesting structure of a program.

Examples

```
PROC_ONE

REC-1-LEN

P OF QTEST-5 OF TEST-4 OF TEST-3
```



Qualification	Item Referred to by Qualification
P	2
P OF P	3
P OF P OF P	4
P OF Q	8
P OF O.B.	2
R	9
S	11
P OF R OF Q OF O.B.	10
P OF R	10
S OF Q	none

Figure 2–1. Subprogram Nesting Structure

The program shown in Figure 2–1 declares one external subprogram R (subprogram R is declared in subprogram Q that resides in the host), plus the structure of the separately compiled subprogram R. As R is bound into the host, all subprogram identifiers become applicable to subprograms nested within R as well as to those subprograms initially residing in the host. Each subprogram is given a number in the example so that it can be uniquely identified *O.B.* is the name given to the unnamed outer block so that it can be used as a qualifier.

Section 3
Binder Statements

Binder statements let you initiate certain binding operations or specify file names or identifiers to be used in the binding process.

You specify Binder statements in the primary input file in free form on one or more records. Place a semicolon (;) after each statement.

A percent sign (%) appearing in any column from 1 through 72 of a record directs Binder to ignore the remaining columns of the record. Binder automatically ignores columns 73 through 80.

An example of a Binder primary input file within a WFL job is shown below. The WFL job is indicated in bold type.

```
? BEGIN JOB BIND/RESULT;
  BIND COBOL74/EXAMPLE BINDER;
  BINDER DATA;
  HOST IS COBOL74/HOST;
  USE S1 FOR PROG;
  BIND S1 FROM COBOL74/PROG;
  STOP;
? END JOB.
```

The various Binder statements are shown in Table 3–1. The syntax and examples for each statement are provided in this section.

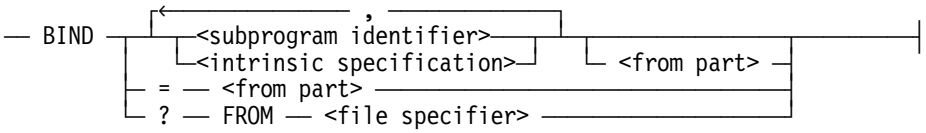
Table 3–1. Binder Statements

Statement	Description
BIND	Indicates the name of a subprogram or intrinsic to be bound, the title of the file containing the subprogram or intrinsic, or both.
DONTBIND	Directs Binder not to bind a specified subprogram.
EXTERNAL	Nonpreferred synonym for DONTBIND.
HOST	Indicates the name of the host program to which a subprogram will be bound.
INITIALIZE	Specifies the address couple of a Master Control Program (MCP) global item for intrinsic binding.
PURGE	Causes the file or group of files indicated by the file specifier to be removed from disk after binding.
STOP	Indicates the end of the primary input file.
USE	Matches the identifiers of external subprograms with the identifiers in the host program.

BIND Statement

Use the BIND statement to specify the name of a subprogram or intrinsic to be bound, the title of the file where the subprogram or intrinsic can be found, or both.

Syntax



<from part>

Explanation

For an explanation of the metatokens in the preceding syntax diagram, see Section 2.

Details

Using the *BIND...FROM...* Form of the BIND Statement

With the *BIND <subprogram identifier> FROM <file specifier>* construct, you can specify only one subprogram identifier, unless the file specifier names a library file (an ALGOL, FORTRAN, or FORTRAN77 program compiled with the LIBRARY option set to TRUE).

Replacement Binding

You can use the BIND statement to bind in a new version of a subprogram already bound to a host. This action is called *replacement binding*.

(Replacement binding is not possible with a C program. Refer to “C Intralanguage Binding” in Section 4 for more information.)

Binding External Subprograms

It is not necessary to use a BIND statement to declare external subprograms (those not already bound to the host). Rather, you can declare the subprograms as external in the host program, and use the *BIND* = form of the BIND statement to indicate the file that contains the subprograms to be bound.

Using the *BIND* = Form of the Bind Statement

You should use the *BIND* = construct when a subprogram identifier is specified in a BIND statement and no file is provided. Binder attempts to locate the file containing the subprogram by using the directory names declared in this construct. Binder replaces the equal sign (=) with the name of the subprogram to be bound.

You can supply only one *BIND* = statement to Binder. If you supply more than one, only the last statement is used.

If a *BIND* = statement is required for proper binding, but you did not include that statement in the CARD input file, Binder creates a *BIND* = statement by using the host program title and substituting the last identifier with an equal sign. For example, if the host program has the title THIS/IS/MY/HOST, Binder creates the following statement:

```
BIND = FROM THIS/IS/MY/=
```

After creating a *BIND* = statement, Binder replaces the equal sign with the subprogram name and looks for a file with that title. For example, if PF is the subprogram to be bound, the BIND statement assumes the following form:

```
BIND PF FROM THIS/IS/MY/PF
```

Binder looks for subprogram PF in the file titled THIS/IS/MY/PF and binds the subprogram if it is found.

During intrinsic binding when no host program is used, Binder creates the *BIND* = statement by substituting the code file title for the host program title.

Using the *BIND ?* Form of the BIND Statement

The *BIND ? FROM <file specifier>* form of the BIND statement is used to bind programs written in C. In place of <file specifier>, you can name either a host program or a subprogram written in C. This form of the BIND statement can also be used for non-C subprograms.

If <file specifier> names . . .	Then the BIND ? FROM statement . . .
A host program written in C	Is interpreted as *HOST IS . . ..* For example, if you specify a host program named TEST, Binder interprets the *BIND ? FROM TEST* statement as *HOST IS TEST*.
A subprogram written in C	Binds all functions and data objects in the specified file, even if the file does not provide a function referenced by the host or other suprograms. Note that you can bind a C subprogram only to a C host program.
A subprogram not written in C	Replaces the question mark in the BIND ? FROM statement with the names of all functions exported from the file. For example, if object file F has functions X, Y, and Z, then the statement BIND ? FROM F is the same as BIND X, Y, Z FROM F.

Examples

The following example binds subprogram SUBA with subprogram P, which is nested in subprogram Q. For details on subprogram nesting structure and restrictions, refer to "Subprogram Identifier" in Section 2.

```
BIND SUBA, P OF Q
```

The following example directs Binder to bind subprograms SUBA and SUBB from an ALGOL library file labeled ALGOL/LIBFILE.

```
BIND SUBA, SUBB FROM ALGOL/LIBFILE
```

The following example directs Binder to bind subprogram SUBA from the file A/B/C.

```
BIND SUBA FROM A/B/C
```

The following example directs Binder to bind subprograms SUBA, SUBB, and SUBC from the files A/SUBA, A/SUBB, and A/SUBC, respectively.

```
BIND SUBA, SUBB, SUBC FROM A/=
```

The following example directs Binder to look for subprogram SUBA first in TEST1/SUBA, and then in TEST2/SUBA. Then Binder looks for subprogram SUBB first in TEST1/SUBB, and then in TEST2/SUBB.

```
BIND SUBA, SUBB FROM TEST1/=, TEST2/=
```

The following example directs Binder to bind subprogram SUBA nested in SUBX, and subprogram SUBB nested in SUBY, from the file TEST/FILE.

```
BIND SUBA OF SUBX, SUBB OF SUBY FROM TEST/FILE
```

In the following example, Binder looks in the file, THISFILE, for any external subprograms and for any subprograms declared in a BIND statement without a corresponding file specifier. Binder replaces the equal sign with the name of the subprogram.

```
BIND = FROM THISFILE
```

Assuming that there is an external subprogram labeled SUBA, the following example directs Binder to look for SUBA first in A/SUBA, next in B/SUBA, and then in C/SUBA.

```
BIND = FROM A/=,B/=,C/=
```

If the subprogram is SUBA, the following example directs Binder to look for SUBA in SUBA.

```
BIND = FROM =
```

The following examples illustrate how you can specify program files by using the *BIND <subprogram identifier>* and *BIND* = constructs. Each of the three statement groups has the same net effect.

```
BIND = FROM FILEID/=;
BIND P;
BIND Q;
BIND R;
```

```
BIND P,Q,R FROM FILEID/=;

BIND P FROM FILEID/P;
BIND Q FROM FILEID/Q;
BIND R FROM FILEID/R;
```

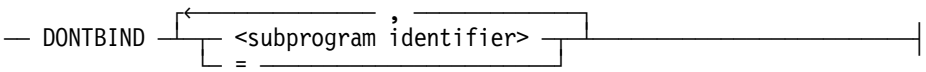
If the subprograms, P, Q, and R were external to the host, they would be bound by default. Thus the following statement would have the same effect as the statements in the previous three examples:

```
BIND = FROM FILEID/=;
```

DONTBIND Statement

Use the DONTBIND statement to direct Binder not to bind a specified subprogram. You can use the DONTBIND statement to suppress the binding of all external subprograms referenced in the host program.

Syntax



Explanation

For an explanation of the metatokens in the preceding syntax diagram, see Section 2.

Details

When you include a subprogram identifier in a BIND statement, but do not include a file specifier, Binder uses the host program title to create a file in which to possibly locate the subprogram. Binder substitutes the subprogram identifier for the last identifier of the host program title and searches for the subprogram in the file under this name. The DONTBIND statement is used to suppress this process.

For example, if a subprogram is declared external to allow it to be invoked by the ALGOL statement, CALL, the DONTBIND statement can be used to suppress the automatic search for the subprogram reference. In addition, the DONTBIND statement can be used if the subprogram is to be bound during a later run of Binder.

Examples

The following example directs Binder to suppress the binding of the subprograms, SUBA and SUB3. The subprograms remain unresolved external references.

```
DONTBIND SUBA, SUB3
```

The following example directs Binder to suppress the binding of all external subprograms referenced in the host program and not explicitly named in a BIND statement. All of the subprograms remain unresolved external references.

```
DONTBIND =
```

Conflicts between BIND and DONTBIND Statements

If multiple BIND and DONTBIND statements apply to a subprogram, Binder selects the statement to use according to the following priority scheme.

1. Binder uses the statement that contains the subprogram identifier with the most qualifiers if the qualification matches the environment of the given subprogram.
2. When a BIND statement and a DONTBIND statement have the same number of qualifiers, Binder uses the BIND statement.
3. When more than one BIND statement applies and each has the same number of qualifiers, Binder uses the last BIND statement. (These rules are referred to in the following paragraphs as priority rule 1, priority rule 2, and priority rule 3.)

In the following example, Binder selects the BIND statement according to priority rule 1.

```
DONTBIND = ;
BIND SUBR;
```

In the next example, Binder selects the BIND statement according to priority rule 2.

```
BIND SUBR;
DONTBIND SUBR;
```

In the following example, potential conflict exists among three BIND statements. If subprogram P is nested in subprogram Q, P is bound from file B/C according to priority rule 1. If subprogram P is not nested in subprogram Q, the statement *BIND P OF Q FROM B/C;* does not apply, and P is bound from file C/D according to priority rule 3.

```
BIND P FROM A/B;
BIND P OF Q FROM B/C;
BIND P FROM C/D;
```

EXTERNAL Statement

Use the EXTERNAL statement to direct Binder not to bind the specified subprogram.

Syntax

Explanation

For an explanation of the metatokens in the preceding syntax diagram, see Section 2.

Details

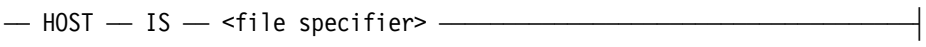
If a subprogram is external to the host, it is left as an unresolved external reference when the EXTERNAL statement is used.

The EXTERNAL statement is the nonpreferred synonym for the DONTBIND statement. Refer to the DONTBIND statement for a more detailed explanation.

HOST Statement

Use the HOST statement to name the title of the host program to which a subprogram is to be bound.

Syntax



Explanation

For an explanation of the metatokens in the preceding syntax diagram, see Section 2.

Details

When the HOST statement is used, any file equation that involves the host program is overridden. If more than one HOST statement appears in the primary input file, only the last HOST statement is effective.

A HOST statement (or host program equation) is not necessary when binding a C program if the host program is specified in a *BIND ?* statement. The file containing the "main" function is implicitly the host program.

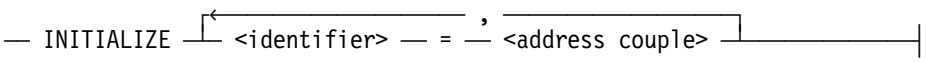
Examples

```
HOST IS MY/HOST;
HOST IS *SYSTEM/PL/I;
HOST IS (MYUSERCODE)HOST/FILE;
```

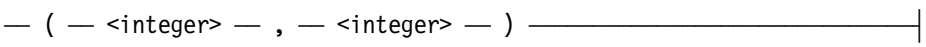
INITIALIZE Statement

Use the INITIALIZE statement when binding intrinsics to specify the correct address couple of an MCP global item.

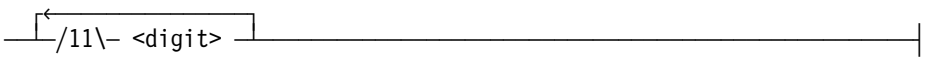
Syntax



<address couple>



<integer>



Explanation

<digit> Any one of the Arabic numerals 0 (zero) through 9.

For an explanation of the other metatokens in the preceding syntax diagram, see Section 2.

Details

This statement is necessary because compilers do not have the correct address couple of the MCP global item at compiling time.

Take extreme care when using the INITIALIZE statement, because unpredictable results can occur.

For details about binding intrinsics, see Section 6.

Examples

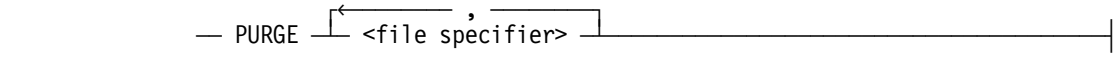
```
INITIALIZE A = (0,50);
INITIALIZE BLOCKEXIT = (0,10);
```

PURGE Statement

The PURGE statement causes the file or group of files indicated by the file specifier to be removed from disk after Binder has finished processing.

Syntax

<purge statement>



Explanation

For an explanation of the metatokens in the preceding syntax diagram, see Section 2.

Details

If a file is specified in the PURGE statement but is not successfully bound, it will not be removed from disk after Binder has finished.

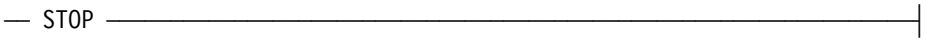
Examples

```
PURGE SEP/FILE;
PURGE =;
PURGE FILE1/=,FILE2/=;
```

STOP Statement

Use the STOP statement to indicate the end of the primary input file. The STOP statement is optional.

Syntax



Details

Binder ignores all records that follow the STOP statement in the input file. This feature allows you to store additional Binder input records in one file without having them executed.

Example

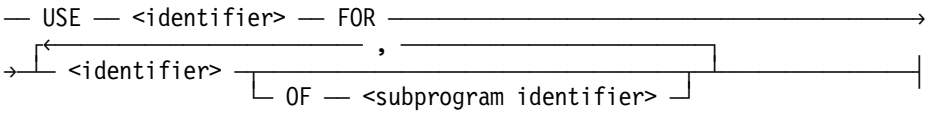
```
STOP;
```

USE Statement

The USE statement directs Binder to match a specified identifier in a subprogram with a specified identifier in a host program. Use the USE statement when identifiers in the subprogram have different names from the identifiers in the host program.

The ALGOL, COBOL, FORTRAN77, NEWP and Pascal languages are not case sensitive; therefore, identifiers are implicitly read as uppercase by these compilers. In the USE statement, uppercase characters must be used to reference identifiers from these languages.

Syntax



Explanation

For an explanation of the metatokens in the preceding syntax diagram, see Section 2.

Details

The first identifier following USE is the identifier contained in the host.

The identifiers following *FOR* are identifiers referenced by the subprograms to be bound into the host. For example, the statement, *USE A FOR B,C,D*, directs Binder to use the host identifier, A, anytime it encounters the subprogram identifiers, B, C, or D.

When a USE statement is invoked and the host identifier does not exist in the host program, the identifier referenced by the subprogram is added to the host program and given the name of the host identifier as specified in the USE statement. The identifier referenced by the subprogram can be qualified by the subprogram identifier so that the USE statement is invoked only when the specified subprogram is bound.

Special considerations are necessary when the USE statement includes the subprogram name itself. According to file-naming conventions, the identifier following the last slash in the subprogram title is the subprogram name. (For more information concerning file-naming conventions, refer to “BIND Statement” in this section.) However, when Binder looks for the subprogram through the directory name, as in OBJECT/LIB/=, it looks for a file title ending with the subprogram name as found in the host program.

If Binder is directed to bind a subprogram from a specified file, and it discovers that the subprogram identifier does not match the subprogram identifier in the host, Binder issues a warning message and creates a USE statement that corrects the identifier mismatch.

Examples

The following examples set up the following correspondences:

- Use the host identifier ALGOLARRAY whenever the COMMON block is referenced in a FORTRAN or FORTRAN77 subprogram.
- Use the host identifier A whenever the subprogram identifiers B, C, or D are encountered.
- Use the host identifier Z whenever the subprogram identifier Y is encountered in the subprogram identified as SUBR2.

```
USE ALGOLARRAY FOR /COMMON/;
USE A FOR B,C,D;
USE Z FOR Y OF SUBR2;
```

For the following two examples, assume that Q is a subprogram contained in a file titled A/Q. If the input to Binder is as shown in the first example, Binder will attempt to bind from file A/P. That is, the USE statement does not cause Binder to bind subprogram P from file A/Q. The BIND statement included in the second example is necessary for correct binding.

```
BIND = FROM A/=;
USE P FOR Q;
BIND P;

BIND = FROM A/=;
USE P FOR Q;
BIND P FROM A/Q;
```

Section 4
Binding Programs Written in the Same Language

The process of binding one or more subprograms to a host written in the same language is known as *intralanguage* binding. This section discusses the various techniques required to perform intralanguage binding for programs written in ALGOL, C, COBOL, FORTRAN, FORTRAN77, and PL/I. You cannot perform intralanguage binding for NEWP and Pascal programs.

ALGOL Intralanguage Binding

Note: *In this section, any reference to ALGOL also refers to the extensions of ALGOL, such as BDMSALGOL, DCALGOL, and DMALGOL.*

ALGOL intralanguage binding consists of binding one or more ALGOL procedures (or subprograms) into an ALGOL host program. The declaration of the subprogram must match its declaration in the host as to the type of subprogram, its number and type of parameters, and its execution level.

Compiling ALGOL Host Programs and Subprogram

An ALGOL host program can be either the outer block of an ALGOL program or an ALGOL procedure.

An ALGOL subprogram compiled independently is called a *separate procedure*. To bind the procedure to a host program, you must compile the procedure at the same lexical level as that of the procedure within the host. Use the Binder control record option, LEVEL, described in Appendix B, to set the desired lexical level of the procedure you want to bind. If you do not set the LEVEL option, the lexical level of the procedure is 3.

Parameter names declared in the procedure need not be the same as those declared in the host. If unmatched identifiers exist in the host and in the procedure you want to bind, declare a Binder USE statement to correct the mismatch. (For the syntax and explanation of the USE statement, see Section 3.)

A separate procedure can reference any item declared in the host that is global to the lexical level of that procedure. This includes items at intermediate lexical levels. However, when the item is an array, and the item is declared in the host at an intermediate lexical

level, the size of the array is always as declared in the host program rather than as declared in the separate procedure.

Any item declared after the body of a given procedure in a host program can be referenced by a separately compiled procedure that replaces the original procedure. Thus, a host program that contains a separately compiled and bound procedure might produce different results from the same program fully compiled by the ALGOL compiler.

Note: *Binder can only perform replacement binding on ALGOL exception and epilog procedures. Only one procedure, epilog or exception can appear in a block.*

Declaring Global Items within an ALGOL Procedure

You must declare all global items within the separate procedure that references them. This ensures that no undeclared global identifiers exist within the procedure. You can declare global items by using either the *brackets* method or the *INFO file* method described later in this section.

Using the Brackets Method

With this method, you enclose global item declarations in brackets and place the declarations before the separate procedure. A bracketed set of global declarations, also known as the *global part*, is illustrated in the following example.

```
[REAL S;
ARRAY B [1];
FILE LINE;
PROCEDURE PROC (V); VALUE V;
REAL V; EXTERNAL;]
```

When a compilation includes multiple procedures, you must include all global items referenced by all procedures within the same set of brackets and place the global part before the first procedure to be compiled.

Following are some Binder exceptions to the typical way of declaring ALGOL items:

- You can declare an array with lower bounds only.
- You can declare switch items without declaring the corresponding switch list items.
- You cannot declare LABEL items, as new global items unless a “bad GO TO” to that item appears in the host. (A “bad GO TO” transfers control from an inner block to an item that is global to that block.)
- If an array declared in the outermost block of the host program and the matching global array in the subprogram are different sizes, and the host array is not an equivalence array, the array in the bound code file is the larger of the two arrays.
- If an array declared in an intermediate level, but not the outermost block, of the host file is matched to a global array in a subprogram, the array in the bound code file is the one declared at the intermediate level in the host program, not the one declared in the subprogram.

- If a global array in the subprogram is bound to an equivalence array in the host program, the array in the bound code file takes on the size of the array in the ALGOL host program.

Using the INFO File Method

With this method of declaring global items, you store declared information for Binder in an INFO file. You can create an INFO file by placing the DUMPINFO compiler control record at any point within the symbolic file of the host program and compiling your ALGOL program with DUMPINFO set to TRUE.

DUMPINFO places information about all items within the scope of the DUMPINFO control record into the INFO file. For example, if a DUMPINFO compiler control record is placed just before the last END statement of a program, all global items declared in that program are described in the INFO file.

To recover the information about the declared items from the INFO file, include the LOADINFO compiler control record in the subprogram before the first procedure to be compiled.

Note that INFO files created by the ALGOL compiler cannot be used by an ALGOL compiler of a different release level. For example, if an INFO file is created by the DUMPINFO compiler control option of the Mark 3.8 ALGOL compiler, the INFO file cannot be used by the LOADINFO compiler control option of the Mark 3.9 ALGOL compiler. If the release levels of the INFO files do not match, a syntax error message is given and the compilation is discontinued.

For a description of INFO files and the DUMPINFO and LOADINFO compiler control options, refer to the *ALGOL Reference Manual*, *Volume 1*.

You can use a combination of the INFO file and brackets methods to add global items to the host without recompiling the host program. To do this, place the LOADINFO compiler control record within the brackets before the first global declaration.

Adding New Global Items to an ALGOL Host Program

If a subprogram references a global item not declared in the host program, Binder adds the item to the host as a *new* global item when binding the procedure into the host. Binder adds new global items at the global level of the host.

The following rules apply when Binder adds new global items to a host program:

- Binder cannot add the following variable types as new global items:
  - DMSII database
  - FORMAT
  - LABEL
  - LIBRARY
  - LIST

  - PICTURE
  - SDF form record libraries
  - SIM databases
  - STRING
  - Switch items
  - Transaction base
  - TRUTHSET
  - TRANSLATETABLE
  - VALUE ARRAY
- Binder can add a new global array only if the array is declared in the subprogram with both upper bounds and lower bounds.
- Binder strips a new global file of any specified file attributes during the binding process. Thus, you must indicate all necessary file attributes by using a file equation. Note that if a global file is already present in the host and is being replaced during the binding procedure, the file attributes specified in the host are used.

Using the ALGOL Separate Compilation Facility

The ALGOL compiler provides a separate compilation and binding facility called *sepcomp*. The sepcomp facility lets you easily recompile and bind procedures contained within one large symbolic file. When you make changes to a procedure, you can use the sepcomp facility to recompile only the changed procedure, rather than recompiling the entire program.

To use the sepcomp facility, you must first create a host object code file by compiling the program with the ALGOL MAKEHOST compiler control option set to TRUE. This causes the ALGOL compiler to save special Binder information in the object code file.

Next, make changes to your program in a patch file. The first record of the patch file must set the ALGOL SEPCOMP compiler control option to TRUE. This signals the compiler to perform a separate compilation.

During the sepcomp process, the ALGOL compiler determines which procedures are affected by the patch file and recompiles those procedures. The compiler then invokes Binder to bind the recompiled procedures into the rest of the object code obtained from the host file.

(Refer to the *ALGOL Reference Manual*, *Volume 1* for more information about the SEPCOMP and MAKEHOST options.)

Library Binding in ALGOL

You can bind libraries and library objects like other locally or globally declared nonprocedure items. You can declare exported procedures as EXTERNAL, and you can bind and replacement bind them. For more information about libraries and exported

procedures, refer to the *System Software Utilities Manual* and the *ALGOL Programming Manual, Volume 1*.

Libraries in subprograms do not have to be explicitly declared in the host program. If libraries are not declared in the host program, Binder builds a library template from the binding information in the subprogram file. Once the template is built, Binder can add library objects not explicitly declared in the host program. When declaring libraries in the global part, you must declare the library before declaring the library object.

The restrictions that apply to library binding are as follows:

- Additional library objects can be added to a library declared in the host program if the host program was compiled with a Mark 3.8 compiler or a more recent version of the compiler.
- Library attributes cannot be changed or added from a subprogram. The library attributes in the host are always used, so it is not necessary to include any attributes in the subprogram.
- The declaration of a procedure to be bound to an exported procedure must be identical to the declaration of the exported procedure.
- Library objects and *by-calling* procedures cannot be declared external and bound in. (A by-calling procedure is a procedure that is declared in a library program and is specified to be exported dynamically.)
- By-calling procedures cannot be declared in the global part of a procedure that is to be bound to a host program.
- If a new global library object is declared in two or more subprograms, then the global library objects must be identical and match the library object in the library referenced.

Record Binding in ALGOL

Binding of records retrieved from a data dictionary is allowed in ALGOL. However, Binder does not check the format of the records involved, nor does it make any distinction between records and EBCDIC arrays. Thus, Binder allows any record to be bound to any other record, and allows any record to be bound to any star-bounded EBCDIC array and vice versa.

Example of ALGOL Intralanguage Binding

The following example shows an ALGOL host program, a subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

ALGOL Host Program

```
? BEGIN JOB ALGOL/HOST;
  COMPILE OBJECT/HOST ALGOL LIBRARY;
  ALGOL DATA
  BEGIN
    FILE LINE(KIND=PRINTER,MAXRECSIZE=22);
    ARRAY BUFFER[0:2,0:8];
```

```
    REAL J;
    PROCEDURE PRINTIT; EXTERNAL;
    FOR J := 0 STEP 1 UNTIL 8 DO
BUFFER[0,J] := BUFFER[1,J] := BUFFER[2,J] := " ";
    FOR J := 0 STEP 1 UNTIL 2 DO
BUFFER[J,1] := BUFFER[J,3] := BUFFER[J,5] := "*";
   BUFFER[1,2] := "*";
    PRINTIT;
  END.
? END JOB.
```

ALGOL Subprogram

```
? BEGIN JOB COMPILE/PRINTIT;
  COMPILE OBJECT/PRINTIT ALGOL LIBRARY;
  ALGOL DATA
  [ARRAY BUFFER[0,0]; REAL J, K; FILE LINE;]
  PROCEDURE PRINTIT;
  BEGIN
    FOR J := 0 STEP 1 UNTIL 2 DO
        WRITE(LINE,<3A1,X1,3A1>, FOR K := 1 STEP 1 UNTIL 6
                                 DO BUFFER[J,K]);
  END;
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/PRINTIT;
  BIND OBJECT/HELLO BINDER LIBRARY;
  BINDER DATA
  HOST IS OBJECT/HOST;
  BIND PRINTIT FROM OBJECT/PRINTIT;
  STOP;
?  END JOB.
```

The result of the bind is a program entitled OBJECT/HELLO. When OBJECT/HELLO is executed, it produces the following output:

```
* *  *
***  *
* *  *
```

Example of Binding an ALGOL Library

This binding example includes an ALGOL library host program, an ALGOL subprogram to be bound to that library, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

ALGOL Host Program

```
? BEGIN JOB COMPILE/LIB/HOST;
  COMPILE OBJECT/LIB/HOST ALGOL LIBRARY;
 ALGOL DATA
  BEGIN
      PROCEDURE REVERSE (A,B,LEN);
          VALUE LEN;
          EBCDIC ARRAY A,B [0];
          INTEGER LEN;
      BEGIN
      END;

      EXPORT REVERSE;
      FREEZE (TEMPORARY);
  END.
? END JOB.
```

ALGOL Subprogram

```
? BEGIN JOB COMPILE/REVERSE;
  COMPILE OBJECT/REVERSE ALGOL LIBRARY;
  ALGOL DATA
  PROCEDURE REVERSE (A,B,LEN);
      VALUE LEN;
      EBCDIC ARRAY A,B [0];
      INTEGER LEN;
  BEGIN
      INTEGER J;
      IF LEN > 0
         AND LEN <= SIZE(A) AND LEN <= SIZE(B)
      THEN
         FOR J := 0 STEP 1 UNTIL (LEN-1) DO
            REPLACE B[J] BY A[LEN-J-1] FOR 1;
  END;
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/SYSTEM/MYLIB;
  BIND SYSTEM/MYLIB BINDER LIBRARY;
  BINDER DATA
  HOST IS OBJECT/LIB/HOST;
  BIND REVERSE FROM OBJECT/REVERSE;
```

```
  STOP;
? END JOB.
```

The result of the bind is a library named SYSTEM/MYLIB. This library is used in the following example.

Example of Binding an ALGOL Program That References a Library

This example shows an ALGOL host program, a subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type. This example uses the library, SYSTEM/MYLIB, created in the preceding example.

ALGOL Host

```
? BEGIN JOB COMPILE/HOST;
  COMPILE OBJECT/HOST ALGOL LIBRARY;
  ALGOL DATA
  BEGIN
      LIBRARY L (LIBACCESS = BYTITLE,
                 TITLE = "SYSTEM/MYLIB.");
      PROCEDURE REVERSE (A,B,LEN);
          VALUE LEN;
          EBCDIC ARRAY A,B [0];
          INTEGER LEN;
      LIBRARY L;
      PROCEDURE P; EXTERNAL;
      P;
  END.
? END JOB.
```

ALGOL Subprogram

```
? BEGIN JOB COMPILE/P;
  COMPILE OBJECT/P ALGOL LIBRARY;
  ALGOL DATA
  [LIBRARY L (LIBACCESS = BYTITLE,
              TITLE = "SYSTEM/MYLIB.");
   PROCEDURE REVERSE (A,B,LEN);
       VALUE LEN;
       EBCDIC ARRAY A,B [0];
       INTEGER LEN;
   LIBRARY L;
  ]
  PROCEDURE P;
  BEGIN
      EBCDIC ARRAY E1,E2 [0:29];
      FILE F (KIND = PRINTER);
      REPLACE E1[0] BY "ABCDEFGHIJKLMNOPQRSTUVWXYZ   ";
      REPLACE E2[0] BY " " FOR 30;
      REVERSE (E1,E2,10);
      WRITE (F,5,E1);
      WRITE (F,5,E2);
  END;
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/P;
  BIND OBJECT/BOUND BINDER LIBRARY;
  BINDER DATA
  HOST IS OBJECT/HOST;
  BIND P FROM OBJECT/P;
  STOP;
? END JOB.
```

The result of the bind is a file named OBJECT/BOUND. When executed, OBJECT/BOUND produces the following result:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
JIHGFEDCBA
```

C Intralanguage Binding

C intralanguage binding involves binding C subprograms into a C host program. Replacement binding is not supported. You can use the *BIND ?* form of the BIND statement to specify the host program and all subprograms. Alternatively, you can use a HOST statement in place of the BIND ? statement to specify the host program.

C Host Programs

A C host program is a C program that contains the function, "main". The host program can contain variables and functions that are referenced by external functions.

You cannot use a bound C program as a host program for a subsequent bind.

If the host program appears in a *BIND ?* statement (see Section 3), it is not necessary to file equate the host program or use a HOST statement in the primary input file.

C Subprograms

A C subprogram file is any C code file that does not contain the function, "main". Unlike with other programming languages, you are not required to use an explicit compiler control option, such as LEVEL, to create a C subprogram file. You can, however, create a level-3 program to bind into a C host program by compiling a C program that includes a "main" function with the compiler option LEVEL set to 3 (LEVEL=3). For more information on binding this level of C programs, refer to "Binding Level-3 C Programs" later in this section.

Describing Functions and Global Variables

Any function or global variable not declared with the storage class specifier, *static*, is implicitly exported.

All references to a global variable or function must match the declaration of the global variable or function in type, number, and types of parameters.

If the bound program is to be a library, you must define all the library entry points in the host program. A subprogram cannot add entry points.

Binding with Different Memory Models

Subprograms compiled with any memory model can be bound to a host compiled with the TINY or LARGE memory model if the FARHEAP compiler control option is set. Subprograms compiled with the LARGE or HUGE memory model can be bound to a host compiled with the LARGE or HUGE memory model. The number of initial far heap rows in the bound C program is determined by the maximum number of initial rows required by the memory models of the host and the subprograms. The initial size of the near and far heap rows is determined by the maximum heap row size required by the memory models

of the host and the subprograms. Use of near or far heap by the host or a subprogram depends on its own memory model. The following table illustrates the size of the heap that results from cross binding different memory models.

Table 4–1. Heap Size in Bound Code Files

	Tiny Subprogram	Small Subprogram	Large Subprogram	Huge Subprogram
Tiny Host	Tiny Heap	Small Heap	Large Heap	Huge Heap
Small Host	Small Heap	Small Heap	Huge Heap	Huge Heap
Large Host	Bind Error	Bind Error	Large Heap	Huge Heap
Huge Host	Bind Error	Bind Error	Huge Heap	Huge Heap

The binder does not issue an error for an incompatibly typed pointer parameter to a function. Use function prototypes to allow the C compiler to detect incompatible pointer parameters. If near and far type qualifiers are not used in function prototypes, the function being called needs to determine whether an actual pointer parameter is near or far. If near and far type qualifiers are used in the function prototypes contained in a header file, a C subprogram that includes this header is able to call the functions without casting the actual pointer parameters, if no attempt is made to pass a far pointer to a near pointer. The other cases (near to near, near to far, and far to far) are all compatible.

Example of C Intralanguage Binding

The following example shows a C host program, a C subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

C Host Program

```
? BEGIN JOB C/MAIN;
  COMPILE C/MAIN CC LIBRARY;
  CC DATA
    #include <stdio.h>
    extern int add (int x, int y);
    print_it (int s)
      {printf ("the sum is %d\n", s);}
    main ()
      {int i = 12, j = 24;
       add (i, j);}
? END JOB.
```

C Subprogram

```
? BEGIN JOB C/ADD;
  COMPILE C/ADD CC LIBRARY;
```

```
  CC DATA
    extern_print it (int s);
    int add (int x, int y)
      {print_it (x + y);
       return x + y;}
? END JOB.
```

Binder Input File

```
? BEGIN JOB C/BIND;
  BIND SUM BINDER;
  BINDER DATA
  BIND ? FROM C/ADD;
  BIND ? FROM C/MAIN;
? END JOB.
```

The result of the bind is a file named SUM. When executed, SUM produces the following output:

```
the sum is 36
```

Binding Level-3 C Programs

You must bind a level-3 C program as a single function into a C host program. The external declaration you use to call a level-3 C program must match the declaration of "main" in the level-3 program. Calling the function causes the level-3 program to initialize and run.

External variables and functions in the level-3 program that are not resolved within the level-3 program itself are mapped by Binder to variables and functions at level 2. The level-3 program uses the heap at level 2, so data pointers can be shared. However, a function pointer is usable only in the program where it is created. A level-3 program can make a pointer to an external level-2 function, but using the pointer outside of the level-3 program produces indeterminate results.

To bind a level-3 program, use the BIND statement in the following format:

```
BIND <identifier> FROM <file specifier>;
```

For the identifier, use the name from the external function declaration used to access the level-3 program.

Example of Binding a Level-3 C Program

The following example shows a C host program, a level-3 C program, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

C Host Program

```
? BEGIN JOB C/HOST;
  COMPILE OBJECT/C/HOST CC LIBRARY;
  CC DATA
    extern nestedProgram ();
    int global;
    main ()
      {nestedProgram ();
       nestedProgram ();}
? END JOB.
```

Level-3 C Program

```
? BEGIN JOB C/LEVEL3;
  COMPILE OBKECT/C/LEVEL3 CC LIBRARY;
  CC DATA
$$  set level 3
    extern int global;
    int local;
     main ( )
      {static int staticLocal;
       global++;                /* Adds 1 to global on each call     */
       local++;                 /* Sets local to 1, because local
                                   is initialized on each call       */
       staticLocal++;           /* Sets staticLocal to 1, like local */
       return;}
? END JOB.
```

Binder Input File

```
? BEGIN JOB C/BIND;
  BIND OBJECT/C/BIND BINDER;
  BINDER DATA
      BIND ? FROM OBJECT/C/HOST;
      BIND nestedProgram FROM OBJECT/C/LEVEL3;
? END JOB.
```

COBOL Intralanguage Binding

COBOL intralanguage binding consists of binding a COBOL subprogram into a COBOL host. The declaration of the subprogram must match its declaration in the host as to the type of subprogram, its number and type of parameters, and its execution level.

Note: *COBOL68, COBOL74, and COBOL85 programs usually are bound similarly and, therefore, are discussed in this section generically as COBOL programs. When a difference exists, the version is specified.*

Compiling COBOL Host Programs and Subprograms

A COBOL host program is a COBOL program compiled at the default lexical (lex) level of 2.

You must compile subprograms at a lex level compatible with their usage in the host program. All subprograms must be compiled at one lex level higher than the lex level of the subprogram in which they are declared. For example, a subprogram to be bound to a lex level 3 subprogram must be compiled at lex level 4.

If a subprogram is declared directly in the host, you must set the LEVEL option to 3 during the compilation of that subprogram.

You must describe in the subprogram any parameters being passed to the subprogram from a host program. You must also specify such parameters following the keyword USING in the PROCEDURE DIVISION heading. You must declare parameters in the LOCAL-STORAGE (LD entry) of the DATA DIVISION and identify them as being passed by reference or by content.

Binding an External Procedure to a COBOL Host Program

To bind an external procedure to a COBOL host program, you must declare the procedure as external in the DECLARATIVES portion of the PROCEDURE DIVISION of the host program.

You can indicate the title of the code file containing the subprogram by using the *CODE FILE TITLE IS <mnemonic name>* option within the SPECIAL-NAMES paragraph. Note that using a BIND statement overrides the SPECIAL-NAMES paragraph.

Activating Bound Subprograms

You can activate bound subprograms by using either the ENTER or CALL verb. Immediately following the verb, you must indicate the name of the section in the DECLARATIVES portion that contains the procedure description of the subprogram. Binder uses the section name that declares the subprogram as external to process the external subprogram. All Binder statements that pertain to an external subprogram must reference the corresponding section name.

Global Declarations in Subprograms

Any variable that can be passed or received as a parameter can be declared as a global variable in the subprogram. Untyped procedures, files, and direct files can also be declared as global variables in the subprogram. To declare global variables, use the GLOBAL clause. The following COBOL74 examples illustrate the declaration of global variables in the WORKING-STORAGE SECTION of the subprogram.

```
77 GLASTATUS    GLOBAL   BINARY                  PIC 9(11).
77 BL-EVNT      GLOBAL EVENT.
77 GL-SWFL      INDEX FILE GLOBAL.
01 GL-RCD       RECORD AREA GLOBAL   OCCURS 10    PIC X(180)
01 GL-EBCRAY    GLOBAL.
   03 CMP-ITE   BINARY                           PIC 9(11)
          OCCURS 100  INDEXED BY I1.
```

If most of the variables declared in the WORKING-STORAGE SECTION are global, use the GLOBAL compiler control option. You can set this option throughout the compilation.

The GLOBAL option affects only the variables that are candidates for global declarations and only the items declared in the WORKING-STORAGE SECTION.

You can use the LOCAL or OWN option to override the GLOBAL option. In the following example, items G1, G2, and G3 are declared GLOBAL; I is declared OWN; and L1 is declared LOCAL.

```
$  SET   GLOBAL
 77 G1   BINARY        PIC 9(11).
 77 G2   BINARY        PIC 9(11).
 77 L1   LOCAL
         BINARY        PIC 9(11).
 01 G3.
     03 FLD         BINARY        PIC 9(11) OCCURS 10 INDEXED BY I.
```

Note that Binder strips a new global file of any specified file attributes during the binding process. Thus, you must indicate all necessary file attributes by using a file equation. If a global file is already present in the host and is being replaced during the binding procedure, the file attributes specified in the host are used.

Tasking and Binding

A host program can call a bound subprogram as a task by using either the PROCESS statement or the CALL statement. You can declare data to be shared between the host and the subprogram as global variables. Be aware that 01-level data items with a usage of binary, computational, double, or real might cause run-time errors when passed as parameters to a bound subprogram that is called as a task.

OWN Declarations in the Subprogram

COBOL programs compiled at lex level 3 or higher can declare certain variables as OWN in the subprogram. OWN variables retain their initial values or states throughout repeated exit and reentry of the subprogram in which they are declared.

With the exception of direct switch files, you can declare any item in the WORKING-STORAGE SECTION of the subprogram as OWN by using either the OWN clause or the OWN compiler control option.

All related index names for OWN items are also considered to be OWN. Redefined OWN items are implicitly OWN, so you do not need to specify them in the OWN clause.

If you use the OWN compiler control option throughout the compilation, all variables declared in the WORKING-STORAGE SECTION are declared OWN, unless they are direct files.

You can use the LOCAL or GLOBAL clause to override the OWN option for any individually specified item.

Library Binding in COBOL

Binder lets you bind COBOL programs that are libraries and COBOL programs that reference libraries. You can bind libraries and library objects as you would other locally or globally declared nonprocedure items.

To bind a COBOL library, declare the library as external in the host program and make sure that the procedure parameters match those in the host program.

Libraries in subprograms do not have to be explicitly declared in the host program. If libraries are not declared in the host program, Binder builds a library template from the binding information in the subprogram file. Once the template is built, Binder can add library objects not explicitly declared in the host program. When declaring libraries in the global part, you must declare the library before declaring the library object.

Example of COBOL Intralanguage Binding

The following example contains a COBOL host program and a subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

COBOL Host Program

```
? BEGIN JOB COMPILE/HOST;
  COMPILE COBOL74/HOST COBOL74 LIBRARY;
  COBOL74 DATA
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HOST.
       ENVIRONMENT DIVISION.
```

```
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. A-15.
        OBJECT-COMPUTER. A-15.
        SPECIAL-NAMES.
            "COBOL"/"PROG" IS TO-BE-CALLED.
        INPUT-OUTPUT SECTION.
        FILE-CONTROL.
            SELECT PR ASSIGN TO PRINTER.
        DATA DIVISION.
        FILE SECTION.
        FD PR.
        01 PR-RCD                  PIC X(36).
        WORKING-STORAGE SECTION.
        01 ORIG                    PIC X(36).
        01 NEW                     PIC X(36).
        LOCAL-STORAGE SECTION.
        LD PARMS.
        01 A                       PIC X(36) REF.
        01 B                       PIC X(36) REF.
        PROCEDURE DIVISION.
        DECLARATIVES.
            S1 SECTION.
            USE EXTERNAL TO-BE-CALLED AS PROCEDURE WITH PARMS USING A B.
        END DECLARATIVES.
        THE-MAIN SECTION.
        START.
            OPEN OUTPUT PR.
            MOVE "THIS WILL STOP WHEN THIS LINE ENDS" TO ORIG.
            ENTER S1 USING ORIG NEW.
            WRITE PR-RCD FROM ORIG.
            WRITE PR-RCD FROM NEW.
            STOP RUN.
? END JOB.
```

COBOL Subprogram

```
? BEGIN JOB COMPILE/PROG;
  COMPILE COBOL74/PROG COBOL74 LIBRARY;
  COBOL74 DATA
      $ SET LEVEL = 3
       IDENTIFICATION DIVISION.
       PROGRAM-ID. ARRAY/MIXER.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. A-9.
       OBJECT-COMPUTER. A-9.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 X REF.
           03 ONE        PIC X(5).
           03 SECOND     PIC X(5).
           03 THIRD      PIC X(5).
```

```
            03 FOURTH     PIC X(5).
            03 FIFTH      PIC X(5).
            03 SIXTH      PIC X(5).
            03 SEVENTH    PIC X(5).
            03 EIGHTH     PIC X(1).
        01 Y REF.
            03 FIRS       PIC X(5).
            03 SECON      PIC X(5).
            03 THIR       PIC X(5).
            03 FOURT      PIC X(5).
            03 FIFT       PIC X(5).
            03 SIXT       PIC X(5).
            03 SEVENT     PIC X(5).
            03 EIGHT      PIC X(1).
        PROCEDURE DIVISION USING X Y.
        THE-SUBPROGRAM SECTION.
        MIX.
            MOVE ONE TO SECON.
            MOVE SECOND TO FOURT.
            MOVE THIRD TO FIRS.
            MOVE FOURTH TO THIR.
            MOVE FIFTH TO SIXT.
            MOVE SIXTH TO SEVENT.
            MOVE SEVENTH TO FIFT.
            MOVE EIGHTH TO EIGHT.
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/RESULT;
  BIND COBOL74/EXAMPLE BINDER;
  BINDER DATA;
  HOST IS COBOL74/HOST;
  USE S1 FOR PROG;
  BIND S1 FROM COBOL74/PROG;
? END JOB.
```

The result of the bind is an object file titled COBOL74/EXAMPLE. When executed, the program generates the following output:

```
THIS WILL STOP WHEN THIS LINE ENDS
STOP THIS WHEN WILL ENDS THIS LINE
```

FORTRAN Intralanguage Binding

FORTRAN intralanguage binding consists of binding a FORTRAN subroutine or function into a FORTRAN host.

Compiling FORTRAN Host Programs and Subprograms

A FORTRAN host is a FORTRAN program that contains a main program. The host can also include subroutines or functions compiled with the main program.

A FORTRAN subprogram is a FORTRAN subroutine or function. You must compile the subprogram at a lexical (lex) level consistent with its lex level within the host.

Any subprogram bound into a host must match its invocations in the host program in terms of the number and type of parameters. If the subprogram identifier does not match its invocation as specified in the host, you must use the Binder USE statement to correct the mismatch.

If an entry point is referenced by a host program, but the corresponding subprogram is not referenced, you must use a BIND statement to specify the file in which the entry point is located.

FORTRAN Common Blocks

When a common block is bound, its resulting length is the largest of all the length values declared for that common block in the host and bound subprograms.

Library Binding in FORTRAN

Subroutines and functions can be bound or replacement bound into a host program that references libraries.

Libraries in subprograms do not have to be explicitly declared in the host program. If libraries are not declared in the host program, Binder builds a library template from the binding information in the subprogram file. Once the template is built, Binder can add library objects not explicitly declared in the host program.

Exported subroutines and functions can be replacement bound. You cannot add new exported program units to a host.

You can bind program units that do not reference libraries into host programs that are libraries or that reference libraries.

Example of FORTRAN Intralanguage Binding

The following example shows a FORTRAN host program and subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

FORTRAN Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE FORTRAN/HOST FORTRAN LIBRARY;
      FORTRAN DATA
      DIMENSION ICK(5,55)
      DO 5 I=1,5
      DO 5 J=1,55
5     ICK(I,J)=" "
      DO 10 I=1,55
10    CALL PLOT(ICK,I)
      DO 20 I=1,5
20    WRITE(6,100) (ICK(I,J),J=1,55)
100   FORMAT(1X,55A1)
      CALL EXIT
      END
? END JOB.
```

FORTRAN Subprogram

```
? BEGIN JOB COMPILE/PLOT;
      COMPILE FORTRAN/PLOT FORTRAN LIBRARY;
      FORTRAN DATA
$ SET SEPARATE
      SUBROUTINE PLOT(ICK,I)
      DIMENSION ICK(5,55)
      ICK(3-SIN(I*0.3)*2,I)="*"
      RETURN
      END
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/PLOT;
      BIND SINE BINDER;
      BINDER DATA
      HOST IS FORTRAN/HOST;
? END JOB.
```

The result of the bind is an object file titled SINE. When executed, this program produces the following output:

```
 *******             *******             *******
*       **          **       **          **       **
          **       *           **       *           **
            *******             *******             *
```

FORTRAN77 Intralanguage Binding

FORTRAN77 intralanguage binding consists of binding a FORTRAN77 subprogram into a FORTRAN77 host.

Compiling FORTRAN77 Host Programs and Subprograms

A FORTRAN77 host is a program that has the BINDINFO compiler control option set. If a main program does not exist in the host program, the main program is assumed to be external. A FORTRAN77 host is compiled at an outer lexical (lex) level of 2. The main program is compiled at a lex level of 3.

A FORTRAN77 subprogram can be a FORTRAN77 main program, subroutine, function, or block data subprogram. Each subprogram is compiled at a lex level of 3.

Any subprogram bound into a host must match its invocations in the host in terms of number and type of parameters. If the subprogram identifier does not match its invocation as specified in the host, you must declare a Binder USE statement to correct the mismatch.

In cases where an entry point is referenced by a host but the corresponding outer subprogram is not referenced, you must use a BIND statement to specify the code file in which the ENTRY statement is located.

A FORTRAN77 main program can be replaced by another FORTRAN77 main program or a FORTRAN77 subroutine that has no parameters.

Files

At execution time, file declarations in the host apply to all subprograms that are bound into the host. If a new file is bound into the host, the first declaration for the new file encountered during binding takes precedence over other declarations.

Common Blocks

When a common block is bound, its resulting length is the largest of all the length values declared for that common block in the host and subprograms, as long as the compiler control option CODEFILEINIT is not set in the host. Any common block that has been initialized in the code file cannot be extended.

Library Binding in FORTRAN77

You can replacement bind exported subroutines and functions into a FORTRAN77 host program that is a library. You cannot add or delete exported subprograms from a host program.

Libraries in subprograms do not have to be explicitly declared in the host program. If libraries are not declared in the host program, Binder builds a library template from the binding information in the subprogram file. Once the template is built, Binder can add library objects not explicitly declared in the host program. When declaring libraries in the global part, you must declare the library before declaring the library object.

If you compile a program with the SEPARATE option set, you must declare all libraries and entities imported from those libraries before the first executable program unit. Each program unit can contain references only to those libraries and library objects that it uses.

Example of FORTRAN77 Intralanguage Binding

The following example shows the binding process involved in binding three FORTRAN77 subprograms to a FORTRAN77 host program.

First, the skeleton library program is created and compiled as a Binder host during a Command and Edit (CANDE) session.

File Name: BOUND/LIB/HOST

```
$ SET BINDINFO
      BLOCK GLOBALS
         EXPORT (PASSR, PASSI, EQV)
      END
      SUBROUTINE PASSR (R)
         REAL R
      END
      SUBROUTINE PASSI (I)
         INTEGER I
      END
      LOGICAL FUNCTION EQV (R1, R2)
         REAL R1, R2
      END
      CALL FREEZE ('TEMPORARY')
      END
```

Next, the following three separate subprograms are compiled:

File Name: BOUND/LIB/PASSR

```
$ SET SEPARATE
      SUBROUTINE PASSR (R)
         REAL R
         PRINT *, ' IN PASSR, R = ', R
      END
```

File Name: BOUND/LIB/PASSI

```
$ SET SEPARATE CALLBYREFERENCE
      SUBROUTINE PASSI (I)
         INTEGER I
```

```
      PRINT *, ' IN PASSI, I (ROUNDED) = ', I
   END
```

File Name: BOUND/LIB/EQV

```
$ SET SEPARATE
   LOGICAL FUNCTION EQV (R1, R2)
      PARAMETER (TINYNO = 0.000000001)
      REAL R1, R2
      EQV = ( ABS(R1 - R2) .LT. TINYNO )
   END
```

The following Binder input file is used to bind the three separate code files into the library program:

File Name: BOUND/LIB

```
HOST IS OBJECT/BOUND/LIB/HOST;
BIND PASSR FROM OBJECT/BOUND/LIB/PASSR;
BIND PASSI FROM OBJECT/BOUND/LIB/PASSI;
BIND EQV FROM OBJECT/BOUND/LIB/EQV;
```

The following program references the newly bound library:

File Name: REF/BOUND/LIB

```
   BLOCK GLOBALS
      LIBRARY LIB (TITLE='OBJECT/BOUND/LIB')
   END
   SUBROUTINE PASSR (R)
      REAL R
      IN LIBRARY LIB
   END
   SUBROUTINE PASSI (I)
      INTEGER I
      IN LIBRARY LIB
   END
   LOGICAL FUNCTION EQV (R1, R2)
      REAL R1, R2
      IN LIBRARY LIB
   END
   LOGICAL EQV
   X = 1.7
   Y = 1.7
   CALL PASSR (X)
   CALL PASSI (Y)
   IF ( EQV(X, Y) ) THEN
      PRINT *, ' X STILL EQUALS Y'
   ELSE
      PRINT *, ' X NO LONGER EQUALS Y'
   END IF
   END
```

When executed, the preceding program produces the following output:

```
IN PASSR, R = 1.7
IN PASSI, I (ROUNDED) = 2
X STILL EQUALS Y
```

Usually, when a FORTRAN77 variable or array element appears as an actual argument (that is, the corresponding dummy argument is a variable), the argument is passed by *value-result*. The dummy argument is assigned the value of the actual argument when the subprogram is entered. If the value of the dummy argument changes, the final value of the dummy argument is assigned to the corresponding actual argument when execution of the subprogram is completed. Passing by value-result is also known as *copy-restore*.

Note that in the preceding example, the compiler control option CALLBYREFERENCE is set for PASSI. Therefore, the argument to that subroutine is passed by reference rather than by value-result. Thus, the value of Y is not truncated when Y is passed to an integer.

Suppose that a new version of PASSI, called PASSI2, is created without CALLBYREFERENCE set.

File Name: BOUND/LIB/PASSI2

```
$ SET SEPARATE
      SUBROUTINE PASSI2 (I)
         INTEGER I
         PRINT *, ' IN PASSI2, I (ROUNDED) = ', I
      END
```

A new version of BOUND/LIB binds the new subroutine into the old library.

File Name: BOUND/LIB

```
HOST IS OBJECT/BOUND/LIB;
BIND PASSI FROM OBJECT/BOUND/LIB/PASSI2;
USE PASSI FOR PASSI2;
```

When REF/BOUND/LIB is executed again, the argument to PASSI2 is explicitly truncated to form an integer when the subroutine is entered. The following output is produced:

```
IN PASSR, R = 1.7
IN PASSI2, I (ROUNDED) = 1
X NO LONGER EQUALS Y
```

PL/I Intralanguage Binding

PL/I intralanguage binding consists of binding one or more PL/I subprograms or procedures to a PL/I host. Communication among the procedures is performed through common EXTERNAL declarations within the procedures and through parameters.

Declaring Host Programs and Subprograms

You can declare any external PL/I procedure as a host program if the only parameters declared within the external procedure are CHARACTER VARYING or DECIMAL FIXED.

Any external procedure not specified as the host is considered to be a subprogram. No parameter type restrictions exist for subprograms.

STATIC EXTERNAL Variables

If you declare a STATIC EXTERNAL variable in both the host program and the external procedure, the variable retains the value declared in the host upon binding. If you declare a STATIC EXTERNAL variable only in the external procedure, the variable retains the value declared in the external procedure.

If you declare a STATIC EXTERNAL variable in more than one external procedure but not in the host, a binding error occurs.

Examples of declaring STATIC EXTERNAL variables in a host and in an external procedure are provided in the following example:

```
HOST:PROC;
    DCL A(4) FIXED STATIC EXTERNAL
        INIT (1,2,3,4),
        SEPARATE ENTRY EXTERNAL;
    CALL SEPARATE( );
    PUT DATA (A);
END HOST;
SEPARATE: PROC;
    DCL A(4) FIXED STATIC EXTERNAL INIT(5,6,7,8),
        A1(4) FIXED STATIC EXTERNAL INIT(9,10,11,12);
    PUT DATA (A,A1);
END SEPARATE;
```

When executed, the bound code file produces the following output:

```
A(1)=  1   A(2)=  2   A(3)=  3   A(4)=  4  A1(1)=  9
A1(2)= 10  A1(3)= 11  A1(4)= 12  ;         A(1)=  1
A(2)=  2   A(3)=  3   A(4)=  4   ;
```

You must initialize any STATIC EXTERNAL CONTROLLED or STATIC EXTERNAL BASED variable before using either variable in a declaration. For example, the first set of

code below must be rearranged to initialize and declare the variables in the proper order, as shown in the second set of code.

```
DCL
1 S1(X) STATIC,
    2 A(X) INIT(B(1),B(2),B(3),B(4)),
1 S2 STATIC,
    2 B(2*X) INIT(C(1),C(2),C(3),C(4)),
1 S3 STATIC,
    2 C(2*X) INIT(1,2,3,4),
X STATIC INIT(2);
```

```
DCL
X STATIC INIT(2),
1 S3 STATIC,
    2 C(2*X) INIT(1,2,3,4),
1 S2 STATIC,
    2 B(2*X) INIT(C(1),C(2),C(3),C(4)),
1 S1(X) STATIC,
    2 A(X) INIT (B(1),B(2),B(3),B(4));
```

Variables whose order of declaration causes a program to run incorrectly when bound also cause a level 3 error at compilation time.

Example of PL/I Intralanguage Binding

The following example shows a PL/I host program and subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

PL/I Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE HOST/HOST PL/I LIBRARY;
      PL/I DATA
  HOST:PROC;
      DCL SEPARATE ENTRY (CHAR(*)) EXTERNAL;
      DCL CHR CHAR(8) INIT('ABCDEFGH');
      PUT SKIP LIST (CHR);
      CALL SEPARATE(SUBSTR(CHR,1,3));
  END HOST;
? END JOB.
```

PL/I Subprogram

```
? BEGIN JOB COMPILE/SEPARATE;
      COMPILE SEPARATE/SEPARATE PL/I LIBRARY;
      PL/I DATA
  SEPARATE:PROC(C);
      DCL C CHAR(*);
```

```
        PUT SKIP LIST(C);
    END SEPARATE;
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/SEPARATE;
        BIND BOUND BINDER LIBRARY;
        BINDER DATA
    HOST IS HOST/HOST;
    BIND SEPARATE FROM SEPARATE/SEPARATE;
    STOP;
? END JOB.
```

When executed, the code file, BOUND, produces the following output in the SYSPRINT print file:

```
ABCDEFGH
ABC
```

Section 5
Binding Programs Written in Different Languages

The process of binding one or more subprograms and a host program written in different languages is known as *interlanguage binding*. This section provides information about binding all of the allowable language combinations, which are shown in the following list. Each combination is presented in alphabetical order in this section.

ALGOL-C
ALGOL-COBOL
ALGOL-FORTRAN
ALGOL-FORTRAN77
ALGOL-NEWP
ALGOL-Pascal
COBOL-C
COBOL-FORTRAN
COBOL-FORTRAN77
COBOL-Pascal
FORTRAN-FORTRAN77

Table 5–1 shows the allowable binding combinations.

Table 5–1. Allowable Binding Combinations

Subprogram Language	Host Program Language							
	ALGOL†	**C**	**COBOL**	**FORTRAN**	**FORTRAN77**	**NEWP‡**	**Pascal□**	**PL/I**
ALGOL†	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
C		Yes						
COBOL	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
FORTRAN	Yes	Yes	Yes	Yes	Yes	Yes		
FORTRAN77	Yes	Yes	Yes	Yes	Yes	Yes		
PL/1								Yes

† All references to ALGOL include the various extensions of ALGOL, such as BDMSALGOL, DCALGOL, and DMALGOL.

‡ The NEWP Master Control Program (MCP) can serve only as a host program in binding.

□ Pascal programs can serve only as host programs in binding.

ALGOL-C Interlanguage Binding

ALGOL-C interlanguage binding consists of binding a level-3 ALGOL subprogram into a C host program.

Identifiers

Procedure identifiers can be shared between a C host program and an ALGOL subprogram. Identifiers that name data objects cannot be shared, because a C program stores its external data objects in its heap. For more information about the C program heap, refer to “Accessing the C Heap from ALGOL” later in this section.

The C language is case-sensitive, but the ALGOL language is not. ALGOL identifiers are implicitly read as uppercase by the ALGOL compiler. If a C function identifier contains lowercase letters, you must include a Binder USE statement to map the lowercase characters to uppercase so that ALGOL can access the function.

C Functions

An ALGOL procedure can call standard C functions as long as the functions are bound in from SYSTEM/CC/HEADERS compiled from SYMBOL/CC/HEADERS. Alternatively, you can create a C function that calls the standard function you desire, and then call the C function from the ALGOL procedure.

You can declare C functions with ALGOL linkage, which

- Matches a void C function to an untyped ALGOL PROCEDURE rather than to an INTEGER PROCEDURE, as shown in Table 5–2
- Enables parameters to be passed by reference

ALGOL linkage **is required** for all of the by reference C arguments types that use the “&” character, as shown in Table 5–3. You can use the by-reference types in C programs calling ALGOL programs but not in ALGOL programs calling C programs.

Table 5–2. Corresponding C Function Types and ALGOL Procedure Types

C Function Type	Corresponding ALGOL Procedure Type
int ( ), char ( )	INTEGER PROCEDURE
pointer ( )	INTEGER PROCEDURE
void ( )	INTEGER PROCEDURE
extern"ALGOL"void ( )	PROCEDURE
float ( )	REAL PROCEDURE
long double ( )	DOUBLE PROCEDURE
struct ( ), union ( )	PROCEDURE (RESULT); VALUE RESULT; POINTER RESULT;

Table 5–3. Corresponding ALGOL Parameter Types and C Argument Types

C Argument Type	Corresponding ALGOL Parameter Type
char	INTEGER (by VALUE)
int, pointer	INTEGER (by VALUE)
float	REAL (by VALUE)
long double	DOUBLE (by VALUE)
int&, pointer&	INTEGER (by REFERENCE)
float&	REAL (by REFERENCE)
long double&	DOUBLE (by REFERENCE)
char(&)[ ]	EBCDIC (or ASCII) ARRAY [*]
int(&)[ ], pointer(&)[ ]	INTEGER ARRAY [*]
float(&)[ ], long double(&)[ ],; struct&, union&;	REAL ARRAY [*]
in(&) ( ), char(&) ( ), pointer(&) ( )	INTEGER PROCEDURE
void(&) ( )	INTEGER PROCEDURE
float(&) ( )	REAL PROCEDURE
long double(&) ( )	DOUBLE PROCEDURE
struct(&) ( ), union(&) ( )	PROCEDURE (RESULT); VALUE RESULT; POINTER RESULT;

Pointers

C pointers, including pointers in arrays passed to ALGOL procedures, occur in the C heap in byte units. C pointers of types char and void are passed as parameters and returned from functions in byte units.

Other types of pointers can be passed and returned in units other than bytes depending on the release level of the compiler and the settings of various options.

If an ALGOL procedure uses a pointer to access the C heap, the pointer should be in byte units. If you pass a non-char pointer to an ALGOL procedure, you should declare the argument in C as a void pointer, so that it will be passed in byte units. Likewise, if you write an ALGOL procedure that extracts a pointer from the C heap and returns it as a procedure result, you should declare the ALGOL procedure in the C program with a result type of char* or void* to prevent a unit mismatch.

Parameter Passing

When an ALGOL procedure receives an array parameter from a C program, the array is passed with a lower bound such that indexing the array by zero (0) gives the first element of the C object.

For struct and union typed functions, ALGOL receives a RESULT parameter that points to a place for the procedure to store its result. You must declare any other parameters after the RESULT parameter.

Example of Binding ALGOL Procedures Into a C Host

The following example illustrates the binding of ALGOL procedures into a C host program. The WFL jobs used to compile and bind the programs appear in bold type.

C Host Program

```
? BEGIN JOB C/HOST;
   COMPILE OBJECT/C/HOST CC LIBRARY;
   CC DATA;
    #include <stdio.h>;
    extern "ALGOL" void CALL_C_BACK (void);
    extern "ALGOL" void DISPLAY_C_STRING (char(&)[ ]);
    extern "ALGOL" SUM_C_ARRAY (FLOAT(&)[ ], INT arraySize);
    void C_FROM_ALGOL (void);
      {DISPLAY_C_STRING (ALGOL has called C);}
    main ( );
       {float array [] = {1.1, 2.2, 3.3, 4.4};
       char outBuf [100];
       sprintf (outBuf, "The sum is %f", SUM_C_ARRAY (array, 4));
       DISPLAY_C_STRING (outBuf);
       CALL_C_BACK( );}
? END JOB.
```

ALGOL Subprogram

```
? BEGIN JOB COMPILE/ALGOL;
   COMPILE OBJECT/ALGOL/SUBPROGRAM ALGOL LIBRARY;
   ALGOL DATA;
$$ SET LEVEL 3 LIBRARY;
   [
     Integer procedure C_FROM_ALGOL; external;
   ]
     procedure CALL_C_BACK;
        begin
        C_FROM_ALGOL;
        end;

     procedure DISPLAY_C_STRING (cStr);
           Ebcdic array cStr [*];
        begin
        display (cStr[0]);
        end;

     Real procedure SUM_C_ARRAY (cArray, cArraySize);
           value cArraySize;
           Real array cArray [*];
           Integer cArraySize;
       begin
           Integer index;
       for index ;= cArraySize - 1 step -1 until 0 do
          SUM_C_ARRAY ;=* + cArray[index];
       end.
? END JOB.
```

Binder Input File

```
? BEGIN JOB C/BIND;
   BIND OBJECT/D/BIND BINDER;
   BINDER DATA
     BIND ? FROM OBJECT/C/HOST;
     BIND = FROM OBJECT/ALGOL/SUBPROGRAM;
? END JOB.
```

The result of the bind is a program titled OBJECT/C/BIND that displays the following output;

```
The sum is 11.000000.
ALGOL has called C.
```

Accessing the C Heap from ALGOL

The externally accessible data objects in a C program are stored in an array called a heap. The heap can be accessed as an EBCDIC array or a REAL array. The name and form of the heap depend upon the memory model of the C program and whether the FARHEAP option is TRUE or FALSE, as shown in Table 5–4.

Table 5–4. Name and Format of the C Heap

If the memory model is . . .	And the FARHEAP option is . . .	Then the heap is a . . .	And the EBCDIC array is named . . .	And the REAL array is named . . .
TINY or SMALL	FALSE	One-dimensional array	.Heap_ebcdic To index into the array, use a C pointer.	.Heap To index into the array, divide the C pointer by 6.
LARGE or HUGE	FALSE	Two-dimensional array	.Heap_ebcdic To index into the array with a C pointer, use bits [38;12] of the C pointer for the first dimension and [26;27] for the second dimension.	.Heap To index into the array with a C pointer, divide the C pointer by 6, and then use bits [38;13] for the first dimension and [25;26] for the second dimension.
TINY, SMALL, LARGE, or HUGE	TRUE	Two-dimensional array	.fHeap_ebcdic To index into the array, use bits [38;15] of a C pointer for the first dimension and bits [23;24] for the second dimension.	.fheap_single To index into the array, use bits [38;15] of a C pointer for the first dimension. For the second dimension, divide bits [23;24] by 6.

Example of an ALGOL Subprogram Accessing the C Heap

The following example shows how ALGOL procedures can access the C program heap. The ALGOL global part and the Binder input contain the ALGOL declarations and the Binder USE statements required to access the heap for the different memory models and the FARHEAP option. You can use these ALGOL declarations and Binder USE statements in your own programs. Note that the WFL jobs used to compile and bind appear in bold type.

C Host Program

```
? BEGIN JOB C/HOST;
  COMPILE OBJECT/C/HOST CC LIBRARY;
  CC DATA
    #include <stdio.h>
    extern DISPLAY_C_STRING (char*);
    extern SUM_C_ARRAY (void* array, int arraySize);
    main ()
      {float array [] = {1.1, 2.2, 3.3, 4.4};
       char outBuf [100];
       sprintf (outBuf, "The sum is %f", SUM_C_ARRAY (array, 4));
       DISPLAY_C_STRING (outBuf);}
? END JOB
```

ALGOL Subprogram

```
? BEGIN JOB COMPILE/ALGOL;
  COMPILE OBJECT/ALGOL/SUBPROGRAM ALGOL LIBRARY;
  ALGOL DATA
$$ SET LEVEL 3 LIBARY
  [
    % To select the appropriate declarations for accessing the C
    % heap, add "$$ SET FARHEAP" if the FARHEAP option is set for
    % the C host program; otherwise, add the "$ SET LARGEORHUGE" if
    % the host memory model is LARGE or HUGE.
$$ SET OMIT = FARHEAP OR LARGEORHUGE
    % C heap access for TINY or SMALL memory model without FARHEAP
    Define rowEbcdic  (cPtr)     = 0#,
           colEbcdic  (cPtr)     = cPtr#,
           rowSingle  (cPtr)     = 0#,
           colSingle  (cPtr)     = (cPtr div 6)#,
           heapEbcdic [row, col] = HEAP_EBCDIC_1D [col]#,
           heapSingle [row, col] = HEAP_SINGLE_1D [col]#;
    Ebcdic array                   HEAP_EBCDIC_1D [0];
    Real   array                   HEAP_SINGLE_1D [0];
$$ POP OMIT SET OMIT = FARHEAP OR NOT LARGEORHUGE
    % C heap access for LARGE or HUGE memory model without FARHEAP
    Define rowEbcdic  (cPtr)     = (cPtr.[38:12])#,
           colEbcdic  (cPtr)     = (cPtr.[26:27])#,
           rowSingle  (cPtr)     = ((cPtr div 6).[38:13])#,
           colSingle  (cPtr)     = ((cPtr div 6).[25:26])#,
```

```
            heapEbcdic [row, col] = HEAP_EBCDIC_2D [row, col]#,
            heapSingle [row, col] = HEAP_SINGLE_2D [row, col]#;
      Ebcdic array                  HEAP_EBCDIC_2D  [0, 0];
      Real array                    HEAP_SINGLE_2D  [0, 0];
  $$ POP OMIT SET OMIT = NOT FARHEAP
      % C heap access for any memory model if FARHEAP option is set
      Define rowEbcdic  (cPtr)     = (cPtr.[38:15])#,
             colEbcdic  (cPtr)     = (cPtr.[23:24])#,
             rowSingle  (cPtr)     = (cPtr.[38:15])#,
             colSingle  (cPtr)     = (cPtr.[23:24] div 6)#,
             heapEbcdic [row, col] = HEAP_EBCDIC_FAR [row, col]#,
             heapSingle [row, col] = HEAP_SINGLE_FAR [row, col]#;
      Ebcdic array                  HEAP_EBCDIC_FAR [0, 0];
      Real array                    HEAP_SINGLE_FAR [0, 0];
  $$ POP OMIT
    ]


      Integer procedure DISPLAY_C_STRING (cPtr);
            value   cPtr;
            integer cPtr;
         begin
         display (heapEbcdic [rowEbcdic(cPtr), colEbcdic(cPtr)]);
         end;

      Real procedure SUM_C_ARRAY (cArrayPtr, cArraySize);
            value   cArrayPtr, cArraySize;
            Integer cArrayPtr, cArraySize;
         begin
            Integer start, index;
            Real array reference heapRow [0];
         heapRow := heapSingle [rowSingle(cArrayPtr), *];
         start := colSingle(cArrayPtr);
         for index := start + cArraySize - 1 step -1 until start do
            SUM_C_ARRAY :=* + heapRow[index];
         end.
  ? END JOB.
```

Binder Input File

```
  ? BEGIN JOB C/BIND;
    BIND OBJECT/C/BIND BINDER;
    BINDER DATA
      BIND ? FROM OBJECT/C/HOST;
      USE .Heap_ebcdic  FOR HEAP_EBCDIC_1D;
      USE .Heap         FOR HEAP_SINGLE_1D;
      USE .Heap_ebcdic  FOR HEAP_EBCDIC_2D;
      USE .Heap         FOR HEAP_SINGLE_2D;
      USE .fHeap_ebcdic FOR HEAP_EBCDIC_FAR;
      USE .fHeap_single FOR HEAP_SINGLE_FAR;
      BIND = FROM OBJECT/ALGOL/SUBPROGRAM;
  ? END JOB.
```

The result of the bind is a program titled OBJECT/C/BIND, which displays the following information:

```
The sum is 11.000000.
```

ALGOL-COBOL Interlanguage Binding

ALGOL-COBOL interlanguage binding consists of binding either an ALGOL subprogram into a COBOL host program or a COBOL subprogram into an ALGOL host program.

Table 5–5 matches identifier types between ALGOL and COBOL.

Table 5–5. Corresponding Identifier Types between ALGOL and COBOL

ALGOL	COBOL	COBOL74
REAL ARRAY	COMP OCCURS	REAL OCCURS
INTEGER ARRAY	COMP OCCURS	BINARY OCCURS
BOOLEAN ARRAY	COMP OCCURS	BINARY OCCURS
DOUBLE ARRAY		
COMPLEX ARRAY		
HEX ARRAY	COMP-2	
ASCII ARRAY	ASCII	
EBCDIC ARRAY	DISPLAY	DISPLAY
EBCDIC ARRAY [*] + INTEGER	DISPLAY LOWERBOUNDS + LEVEL 77	DISPLAY LOWER BOUNDS + LEVEL 77
REAL VARIABLE	COMP-4	REAL
INTEGER VARIABLE	COMP or COMP-1	BINARY (1 to 11 digits)
BOOLEAN VARIABLE	COMP or COMP-1	BINARY (1 to 11 digits)
DOUBLE VARIABLE	COMP-5	DOUBLE
COMPLEX VARIABLE		
UNTYPED PROCEDURE	SECTION	SECTION
TYPED PROCEDURE		
UNTYPED PROCEDURE + 2 PARAMETERS (EBCDIC ARRAY [*] + INTEGER)	SECTION + 2 PARAMETERS (DISPLAY LOWERBOUNDS + LEVEL 77)	SECTION + 2 PARAMETERS (DISPLAY LOWERBOUNDS + LEVEL 77)
FILE	FILE	FILE
DIRECT FILE	DIRECT FILE	DIRECT FILE

Global Items

Global items can be shared between COBOL and ALGOL programs. If a COBOL subprogram references a global variable in an ALGOL host program, you must declare the variable in COBOL by using the GLOBAL clause or by setting the GLOBAL compiler control option in the COBOL subprogram to TRUE.

Similarly, when an ALGOL subprogram references a global variable in a COBOL host program, you must declare the variable in ALGOL by using either the brackets method or the INFO file method described in Section 4.

Note that an item declared GLOBAL in a COBOL/COBOL74 program declared at a lexical level greater than 3 can be matched to an ALGOL array declared in a procedure. When the arrays are different sizes, the size declared in the host file is used. However, the larger of the two sizes is used when the array is declared in the outer block of the ALGOL host program.

Note that Binder strips a new global file of any specified file attributes during the binding process. Thus, you must indicate all necessary file attributes by using a file equation. If a global file is already present in the host and is being replaced during the binding procedure, the file attributes specified in the host are used.

Parameters

You must observe the following requirements when passing parameters between ALGOL and COBOL:

- If the ALGOL host program passes arrays to or receives arrays from COBOL subprograms, you must declare the arrays in the ALGOL host program with a lower bound of zero.
- When a word-oriented (integer, real, or Boolean) array or variable is passed between ALGOL and COBOL68, you must declare the word-oriented entity as COMPUTATIONAL in the COBOL68 program.

  In a COBOL74 program, you must declare a real array or variable as REAL, and an integer or Boolean array or variable as BINARY.
- ALGOL EBCDIC arrays correspond to COBOL DISPLAY items.
- You can pass files and direct files between ALGOL and COBOL.
- If a file is declared to have a file record in COBOL, the ALGOL program must declare the file or direct file followed by a *by-value* pointer to match the file record.
- You cannot pass a procedure as a parameter between ALGOL and COBOL.
- Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

Libraries

You do not need to declare attributes for a global library in a procedure to be bound. Instead, Binder uses the attributes found in the host program.

Record Binding

You can bind records retrieved from a data dictionary. However, Binder does not check the format of the records involved, nor does it make any distinction between records and EBCDIC arrays. Thus, Binder allows any record to be bound to any other record, and

allows any record to be bound to any star-bounded EBCDIC array and vice versa. Specifically, this means that you can bind an ALGOL record to a COBOL EBCDIC array.

Binding ALGOL and COBOL74 Programs That Use COMS

ALGOL and COBOL74 use different titles for the COMS support library. ALGOL names the library COMSSUPPORT, while COBOL74 names the library DCILIBRARY.

To prevent library mismatch errors when binding a COBOL74 subprogram to an ALGOL host program, add the following Binder statement to the Binder input file:

```
USE COMSSUPPORT FOR DCILIBRARY
```

When binding an ALGOL subprogram to a COBOL74 host program, add the following Binder statement to the Binder input file:

```
USE DCILIBRARY FOR COMSSUPPORT
```

If you are using COMS service functions, you can avoid possible binding problems by using a naming convention for the library in the COBOL74 subprogram similar to that used in the ALGOL host program. Thus, use an internal name other than DCILIBRARY for the library in which the service functions reside and give the library the function name of COMSSUPPORT, as shown in the following example.

Examples

The following is an example of declaring a COMS service function in ALGOL. For more information on the declaration and use of COMS service functions, see the *ALGOL Reference Manual*, *Volume 2*.

```
LIBRARY SERVICE_LIB
     (FUNCTIONNAME = "COMSSUPPORT", LIBPARAMETER = "02");

INTEGER PROCEDURE GET_NAME_USING_DESIGNATOR
     (ENTY_DESIGNATOR, ENTY_NAME);
          REAL          ENTY_DESIGNATOR;
          EBCDIC ARRAY  ENTY_NAME[0];
LIBRARY SERVICE_LIB;
```

The following is an example of declaring a COMS service function in COBOL74. For more information on the declaration and use of COMS service functions, see the *COBOL ANSI-74 Reference Manual*, *Volume 2*.

```
CHANGE ATTRIBUTE LIBACCESS OF "SERVICE_LIB"
    TO BYFUNCTION.
CHANGE ATTRIBUTE FUNCTIONNAME OF "SERVICE_LIB"
    TO COMSSUPPORT.

CALL "GET-NAME-USING-DESIGNATOR OF SERVICE_LIB"
    USING WS_DESG,
```

```
      WS_NAME
GIVING SF-RSLT.
```

ALGOL-FORTRAN Interlanguage Binding

ALGOL-FORTRAN interlanguage binding consists of binding either an ALGOL subprogram into a FORTRAN host program or a FORTRAN subprogram into an ALGOL host program.

Table 5–6 matches the identifier types between ALGOL and FORTRAN.

Table 5–6. Corresponding Identifier Types between ALGOL and FORTRAN

ALGOL	FORTRAN
REAL ARRAY	REAL ARRAY/COMMON BLOCK
INTEGER ARRAY	INTEGER ARRAY/COMMON BLOCK
BOOLEAN ARRAY	LOGICAL ARRAY/COMMON BLOCK
DOUBLE ARRAY	DOUBLE PRECISION ARRAY/COMMON BLOCK
COMPLEX ARRAY	COMPLEX ARRAY/COMMON BLOCK
HEX ARRAY	
ASCII ARRAY	
EBCDIC ARRAY	
EBCDIC ARRAY [*] + INTEGER	
REAL VARIABLE	REAL VARIABLE
INTEGER VARIABLE	INTEGER VARIABLE
BOOLEAN VARIABLE	LOGICAL VARIABLE
DOUBLE VARIABLE	DOUBLE PRECISION VARIABLE
COMPLEX VARIABLE	COMPLEX VARIABLE
UNTYPED PROCEDURE	SUBROUTINE
TYPED PROCEDURE	FUNCTION
UNTYPED PROCEDURE + 2 PARAMETERS (EBCDIC ARRAY [*] + INTEGER)	
FILE	FILE
DIRECT FILE	

Parameters

When ALGOL and FORTRAN program units are bound, only simple variables, arrays, and labels can be passed as parameters between program units. Procedures, functions, and subroutines cannot be passed as parameters between ALGOL and FORTRAN.

Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

When simple variables and arrays are passed as parameters, the following special conditions apply:

- All FORTRAN arrays are implemented as one-dimensional arrays. Thus, only one-dimensional ALGOL arrays (or array rows) can be passed between ALGOL and FORTRAN program units.
- When passing an ALGOL array to a FORTRAN routine, you must declare the ALGOL array with a lower bound of 0 (zero). The ALGOL subscript 0 (zero) corresponds to the FORTRAN arrays lower bound, usually 1.
- When passing a FORTRAN array to an ALGOL routine, you must declare the ALGOL array with an asterisk (*) for the lower bound. If you use a FORTRAN subscript value to evaluate the parameter, the FORTRAN subscript corresponds to an ALGOL subscript value of 0 (zero). If you do not specify a subscript, the lower bound of the FORTRAN array (usually 1) is used.

  ALGOL describes all simple variable arguments to imported subprograms as call-by-name. FORTRAN describes them as call-by-reference. When calling a library object, Binder allows call-by-reference and call-by-name arguments to match at run time.
- When passing simple variables between FORTRAN and ALGOL, you can mix by-name and by-value parameters. Note, however, that FORTRAN by-value parameters are different from ALGOL by-value parameters. Thus the following conditions apply:
  - If FORTRAN calls an ALGOL procedure and passes a variable as a parameter, the variable acts like an ALGOL by-name parameter in all situations.
  - If FORTRAN calls an ALGOL procedure and passes an expression as a parameter, the expression acts like an ALGOL by-value parameter in all situations.
  - If ALGOL calls a FORTRAN program unit and passes a by-name parameter to a by-value formal parameter, the parameter acts like a FORTRAN by-value parameter.
  - If ALGOL calls a FORTRAN program unit and passes a by-value parameter, the parameter acts like an ALGOL by-value parameter in all situations.

Global Items

The only global items that can be shared between ALGOL and FORTRAN programs are files, subprograms, and common blocks matched to ALGOL arrays. No restrictions are imposed on the referencing of subprograms between the two languages.

Files

An ALGOL file with a declared internal name of *FILEnn*, where *nn* is a 1-digit or 2-digit number from 1 to 99 (without a leading zero), is identified as the same file as a FORTRAN file with that number. Thus, an ALGOL file declaration of *FILE6* and a FORTRAN subprogram file statement of *WRITE (6,1)* refer to the same file.

This also applies to the use of ALGOL files as variable files within a FORTRAN program. For example, assume that an ALGOL host program declares a global file as *FILE12* and declares a FORTRAN subprogram with the following statements:

```
IX=12
READ(IX,7)Y
```

With these statements, the FORTRAN subprogram is bound into the ALGOL host program, and then the ALGOL global file is read.

Note: *The ALGOL print routine performs its carriage control after writing to a printer file. The FORTRAN print routine performs carriage control before writing to a printer file. To prevent potential printing problems, set the WRITEAFTER compiler control option when the ALGOL program is compiled.*

Common Blocks

A FORTRAN common block is a one-dimensional, single-precision array immediately followed by a double-precision copy descriptor. The copy descriptor allows the same data items to be referenced from the single-precision array.

An ALGOL subprogram can access a common block in a FORTRAN host program as a single-precision array, a double-precision array, or both. A common block in a FORTRAN subprogram can access ALGOL single- or double-precision arrays.

When equating ALGOL arrays and FORTRAN common blocks, you must declare the arrays as global.

You must enclose common block names in slashes (/), as in the following example:

```
/ABC/
```

To indicate a blank common block, use the following syntax:

```
/.BLNK./
```

A FORTRAN common block in a subprogram cannot contain an initial value when bound into an ALGOL host array. However, an ALGOL array can be bound to a host common block that contains initial values.

If a FORTRAN common block is equated to an ALGOL host array of a different length, the resulting array in the bound code file will be the longer of the two lengths. However, if the ALGOL array is an equivalence array, the resulting common block will be the size of the array that the equivalence array references.

(Note that an equivalence array is an array that is declared to refer to the same data as another array.)

Array B in the following example is an equivalence array:

```
ARRAY A[0:99];
ARRAY B[1] = A;
```

Simulating Common Blocks in ALGOL

You can determine the contents of a FORTRAN common block by mapping the elements of the common declaration onto a contiguous array. You can simulate this procedure in ALGOL as shown in the following example;

FORTRAN Statements

```
DOUBLE PRECISION DA(10)
COMMON /C/ RA(7),X,DA
```

ALGOL Statements

```
ARRAY C[0:27];
DOUBLE ARRAY D[0] = C;
DEFINE DA(N) = D[(N)+3] #,
       RA(N) = C[(N)-1] #,
       X     = C[7] #;
```

In this example, subscripts are adjusted so that DA(N) and RA(N) in the ALGOL code reference DA(10) and RA(7) in the FORTRAN common block.

Accessing FORTRAN Common Blocks as ALGOL Arrays

The following paragraphs describe how an ALGOL subprogram can access a FORTRAN common block as a single-precision array, a double-precision array, or as both.

Single-Precision Array

To access a FORTRAN common block as a single-precision array, declare a global ALGOL array and equate it to the FORTRAN common block by using the Binder USE statement. For example, the USE statement for a single-precision ALGOL array *A* and a FORTRAN host common block */BLK/* is as follows:

```
USE /BLK/ FOR A;
```

Double-Precision Array

To access a FORTRAN common block as a double-precision array, declare the ALGOL array as double-precision, and then use the USE statement as shown in the previous example for a single-precision array.

Single- and Double-Precision Arrays

To access a FORTRAN common block as both single-precision and double-precision ALGOL arrays, declare both types of ALGOL array and equate the arrays to the FORTRAN common block by using a Binder USE statement.

For example, with a single-precision ALGOL array A, a double-precision ALGOL array D, and a FORTRAN host common block /BLK/, the Binder USE statement is as follows:

```
USE /BLK/ FOR A,D;
```

If the common block and all of the global ALGOL arrays equated to it are of different lengths, the length of the resulting common block will be the longest of all of the lengths. If one of the ALGOL arrays is an equivalence array, the resulting common block will be the size of the array that the equivalence array references.

Accessing ALGOL Global Arrays from a FORTRAN Common Block

The following paragraphs describes how a common block in a FORTRAN subprogram can access single- and double-precision arrays in an ALGOL host program.

Single-Precision Array

To access an ALGOL single-precision array through a FORTRAN common block, equate the array to the common block with a Binder USE statement. For example, given an ALGOL single-precision array A and a FORTRAN common block /BLK/, the Binder USE statement is as follows:

```
USE A FOR /BLK/;
```

Double-Precision Array

To access an ALGOL double-precision array through a FORTRAN common block, you must perform the following steps:

- Declare the double-precision array immediately following the declaration of the single-precision array and equate the double-precision array to the single-precision array by using the equal sign (=).
- Equate the single-precision array to the FORTRAN common block by using a Binder USE statement.

  For example, to access an ALGOL single-precision array A and an ALGOL double-precision array D through a FORTRAN common block /BLK/, you would declare the ALGOL arrays in the ALGOL host program as follows:

```
REAL ARRAY A[0:99];
DOUBLE ARRAY D[0]=A;
```

Then you would use the following Binder USE statement:

```
USE A FOR /BLK/;
```

FORTRAN references to single-precision items in /BLK/ are changed by Binder to refer to array A, and FORTRAN references to double-precision or complex items in /BLK/ are changed to refer to array D. It is not sufficient for array D to be a copy of array A; array D must also be declared immediately following array A.

Example of ALGOL-FORTRAN Binding

The following example illustrates the binding of ALGOL and FORTRAN subprograms into a FORTRAN host program. The WFL job used to compile each program appears in bold type.

FORTRAN Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE FORT/HOST FORTRAN LIBRARY;
      FORTRAN DATA
      DIMENSION X(1), Y(1)
      X(1) = "MENTAT"
      Y(1) = "RESDED"
      WRITE (6,1) X,Y
 1    FORMAT (2(X,A6))
      CALL SUB (X,Y)
      WRITE (6,1) X,Y
      CALL SUBA (X,Y)
      WRITE (6,1) X,Y
      STOP
      END
? END JOB.
```

ALGOL Subprogram

```
? BEGIN JOB COMPILE/FTEST/ALGOL;
      COMPILE FTEST/ALGOL ALGOL LIBRARY;
      ALGOL DATA
      PROCEDURE SUBA(A,B); ARRAY A,B[*];
      BEGIN
         REAL C;
         C ;= A[0];
         REPLACE POINTER (A) BY B[0] FOR 3;
         REPLACE POINTER (B) BY C FOR 3;
      END;
? END JOB.
```

FORTRAN Subprogram

```
? BEGIN JOB COMPILE/FTEST/FORTRAN;
      COMPILE FTEST/FORTRAN FORTRAN LIBRARY;
      FORTRAN DATA
$ SET SEPARATE
      SUBROUTINE SUB(A,B)
      DIMENSION A(1), B(1)
      C = A(1)
      A(1) = B(1)
      B(1) = C
      RETURN
      END
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/ROUND/PROGM;
      BIND ROUND/PROGM BINDER;
      BINDER DATA
      HOST IS FORT/HOST;
      BIND = FROM FTEST/=;
? END JOB.
```

The result of the bind is a program titled, ROUND/PROGM. The execution of ROUND/PROGM generates the following output:

```
MENTAT RESDED
RESDED MENTAT
MENDED RESTAT
```

ALGOL-FORTRAN77 Interlanguage Binding

ALGOL-FORTRAN77 interlanguage binding consists of binding either an ALGOL subprogram into a FORTRAN77 host program or a FORTRAN77 subprogram into an ALGOL host program.

You cannot bind a FORTRAN77 subroutine with a label parameter into an ALGOL host program.

Table 5–7 matches identifier types between ALGOL and FORTRAN77.

Table 5–7. Corresponding Identifier Types between ALGOL and FORTRAN77

ALGOL	FORTRAN77
REAL ARRAY	REAL ARRAY/COMMON BLOCK
INTEGER ARRAY	INTEGER ARRAY/COMMON BLOCK
BOOLEAN ARRAY	LOGICAL ARRAY/COMMON BLOCK
DOUBLE ARRAY	COMMON BLOCK
COMPLEX ARRAY	COMMON BLOCK
HEX ARRAY	
ASCII ARRAY	
EBCDIC ARRAY	CHARACTER COMMON BLOCK
EBCDIC ARRAY [*] + INTEGER	CHARACTER ARRAY/CHARACTER VARIABLE
	DOUBLE PRECISION ARRAY
	COMPLEX ARRAY
REAL VARIABLE	REAL VARIABLE
INTEGER VARIABLE	INTEGER VARIABLE
BOOLEAN VARIABLE	LOGICAL VARIABLE
DOUBLE VARIABLE	DOUBLE PRECISION VARIABLE
COMPLEX VARIABLE	COMPLEX VARIABLE
UNTYPED PROCEDURE	SUBROUTINE/MAIN PROGRAM
TYPED PROCEDURE	FUNCTION
UNTYPED PROCEDURE + 2 PARAMETERS (EBCDIC ARRAY [*] + INTEGER)	CHARACTER FUNCTION
FILE	FILE
DIRECT FILE	

Global Items

The only global items that can be shared between ALGOL and FORTRAN77 programs are subprograms, files, and common blocks. The common blocks map onto ALGOL arrays.

Subprograms

Following are the restrictions to referencing subprograms between ALGOL and FORTRAN77:

- You can replace a FORTRAN77 main program with any ALGOL untyped procedure without parameters by binding a separate file containing an ALGOL procedure to a FORTRAN77 host program. Use the following Binder syntax to indicate the title of the file containing the ALGOL procedure and to indicate the name of the ALGOL procedure to use in place of the main program, .MAIN.:

  ```
  BIND .MAIN. FROM <file specifier>
  USE .MAIN. FOR <identifier>
  ```

- A FORTRAN77 character function can map only onto an ALGOL untyped procedure that has two required leading parameters. The first parameter must be a star-bounded EBCDIC array and the second parameter must be an integer variable. The EBCDIC array maps onto the value of the character function, and the integer variable maps onto the length of the character function.
- Subroutines and functions can be bound or replacement bound into a host program that references libraries.
- New exported subprogram units cannot be added to a host program.
- Libraries can be added to the host program.

Files

An ALGOL file with a declared internal name of FILEnn, where nn is a 1-digit or 2-digit number from 0 to 99 (without a leading zero), is identified as the same file as a FORTRAN77 file with that number. Thus, an ALGOL output statement writing to a file declared globally as FILE6 and a WRITE (6,1) statement in a FORTRAN77 subprogram bound to that ALGOL host program refer to the same file. This method of matching an ALGOL file name with a FORTRAN77 file name also applies to the use of ALGOL files as variable files within a FORTRAN77 program. For example, if the ALGOL subprogram declares a global file FILE12 and writes to it, and the FORTRAN77 host program contains the following statements, they both access the same global file, FILE12:

```
IX=12
READ (IX,7) Y
```

Note: *The ALGOL print routine performs its carriage control after writing to a printer file. The FORTRAN print routine performs carriage control before writing to a printer file. To prevent potential printing problems, set the WRITEAFTER compiler control option when the ALGOL program is compiled.*

Common Blocks

In FORTRAN77, there are two types of common blocks: character and arithmetic. The character common block maps onto an ALGOL EBCDIC array. The arithmetic common block can be accessed by ALGOL as a single-precision array, a double-precision array, or as both.

FORTRAN77, unlike FORTRAN, accesses the common block only through a single-precision descriptor and is not affected by odd offsets.

You must enclose common block names in slashes (/) as in the following example:

```
/ABC/
```

To indicate a blank common block, use the following syntax:

```
/.BLNK./
```

When a host common block is bound, its resulting length is the longest of all the lengths declared for that block in the host program and bound subprograms, provided that the FORTRAN77 CODEFILEINIT compiler control option is not set in the host program. You cannot extend any common block that has been code file initialized. If you attempt an extension, Binder issues the following error message:

```
COMMON BLOCK CANNOT BE EXTENDED BECAUSE IT IS CODEFILE INITIALIZED
```

Accessing FORTRAN77 Common Blocks as ALGOL Arrays

The following paragraphs describe how an ALGOL subprogram can access FORTRAN77 arithmetic and character common blocks as ALGOL arrays.

Single-Precision Array

To access an arithmetic common block as a single-precision array, declare a global ALGOL array and equate it to the FORTRAN77 common block by using the Binder USE statement. For example, with a single-precision ALGOL array titled A and a FORTRAN77 host common block titled /BLK/, the USE statement is as follows:

```
USE /BLK/ FOR A;
```

Double-Precision Array

To access an arithmetic common block as a double-precision array, declare the ALGOL array as double precision and use the same statement as shown in the previous example for a single-precision array.

Single- and Double-Precision Array

To access an arithmetic common block as both single- and double-precision arrays, declare a single- and a double-precision ALGOL array and equate both arrays to the FORTRAN77 common block with the Binder USE statement. For example, with a single-precision array titled

A, a double-precision array titled D, and a host common block titled /BLK/, the USE statement is as follows:

```
USE /BLK/ FOR A,D;
```

EBCDIC Array

To access a character common block as an EBCDIC array, declare a global ALGOL EBCDIC array and equate it to the FORTRAN77 common block by using a Binder USE statement.

Using Initial Values with Common Blocks

You can bind an ALGOL array to a FORTRAN common block that contains data-initialized values. Similarly, you can bind a FORTRAN77 common block containing initial values to an ALGOL host array if the common block is in a main program or a block data subprogram and the FORTRAN77 compiler control option CODEFILEINIT is not set during compilation.

The following are additional restrictions on the use of initial values with common blocks in binding:

- If you plan to data initialize a common block in a given program unit, set CODEFILEINIT to avoid losing the data in the common block if the program unit is ever replaced. The program unit that replaces the original program unit might or might not initialize the common block.
- If you data initialize a common block in one program unit, and then you bind in a different program unit that also initializes the common block, either program unit can supply data for the common block. Thus, the results are unpredictable. For this reason, you should avoid initializing values for the same common block in more than one program unit.

Accessing ALGOL Arrays from a FORTRAN77 Common Block

The following paragraphs describe how to access single-precision and EBCDIC arrays in an ALGOL host program from the common block of a FORTRAN77 subprogram. Double-precision arrays in ALGOL hosts cannot be accessed through FORTRAN77 common blocks.

Single-Precision Array

To access a single-precision array through an arithmetic common block, declare the ALGOL array as a single-precision array and equate the array to the FORTRAN77 common block by using the Binder USE statement. For example,

```
USE A FOR /BLK/;
```

If a common block is equated to an ALGOL array of a different length, the resulting array in the bound code file will be the longer of the two lengths, unless the ALGOL array is an equivalence array. If the ALGOL array is an equivalence array, the resulting common block will be the size of the array that the equivalence array references.

An equivalence array is an array that is declared to refer to the same data as another array. Array B in the following example is an equivalence array.

```
ARRAY A[0:99];
ARRAY B[1] = A;
```

EBCDIC Array

To access an EBCDIC array through a character common block, equate the ALGOL array to the FORTRAN77 common block by using the Binder USE statement.

Simulating Common Blocks in ALGOL

A FORTRAN77 common block is represented internally as a one-dimensional, single-precision array. You can determine the contents of the array by mapping the elements of the common declaration onto a contiguous array. You can simulate this procedure in ALGOL, as shown in the following example:

FORTRAN77 Statements

```
DOUBLE PRECISION DA(10)
COMMON /C/ RA(7),X,DA
```

ALGOL Statements

```
ARRAY C[0:27];
DOUBLE ARRAY D[0] = C;
DEFINE DA(N) = D[(N)+3] #,
       RA(N) = C[(N)-1] #,
       X     = C[7] #;
```

In this example, subscripts are adjusted so that DA(N) and RA(N) in ALGOL reference DA(10) and RA(7) in the common block.

Parameters

When ALGOL and FORTRAN77 program units are bound, only simple variables, arrays, and labels can be passed as parameters between program units. Procedures, subroutines, functions, and intrinsics cannot be passed as parameters between ALGOL and FORTRAN77.

Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

When simple variables and arrays are passed as parameters, the following special conditions apply:

- All FORTRAN77 arrays are implemented as one-dimensional arrays. Thus, only one-dimensional ALGOL arrays (or array rows) can be passed between the two languages.

- When passing a FORTRAN77 array to an ALGOL array, declare the ALGOL array as a star-bounded array.
- If you use a FORTRAN77 subscript value to evaluate the actual parameter, the subscript corresponds to the ALGOL subscript value of 0 (zero).
- If you do not specify a subscript, the FORTRAN77 arrays lower bound is used (usually 1).
- To pass an ALGOL array to a FORTRAN77 subprogram, you must declare the array in ALGOL with a 0 (zero) lower bound (or as star bounded if the array is a formal parameter in the ALGOL subprogram). The specified or default FORTRAN77 lower bound corresponds to the ALGOL 0 (zero) lower bound.
- The following conditions apply to arrays:
  - FORTRAN77 double-precision and complex arrays are implemented as single-precision arrays that need not be even-word aligned.

    Thus, double-precision and complex arrays do not correspond to any ALGOL array and cannot be passed as parameters between the two languages. In some cases, you can override this restriction by using the DOUBLEARRAYS compiler control option described in the *FORTRAN77 Programming Reference Manual*.

  - To pass FORTRAN77 character variables and character arrays to ALGOL, you must use two consecutive formal arguments in the ALGOL subprogram; an ALGOL EBCDIC array with star bounds, and an integer variable.

    The characters are passed with a descriptor, an offset, and a character length. The descriptor and offset correspond to the EBCDIC array argument, and the character length corresponds to the integer variable argument.

- When passing simple variables between FORTRAN77 and ALGOL, you can mix by-name and by-value parameters. Note, however, that FORTRAN77 by-value parameters are different from ALGOL by-value parameters because FORTRAN77 passes noncharacter variables by value-result.

  (Passing by value-result is also known as copy-restore. See “FORTRAN77 Intralanguage Binding” in Section 4 for more information about passing arguments by value-result.)

- When mixing by-name and by-value parameters, the following conditions apply:
  - If FORTRAN77 calls an ALGOL procedure and passes a variable as a parameter, the variable acts like an ALGOL by-name parameter in all situations.
  - If FORTRAN77 calls an ALGOL procedure and passes an expression as a parameter, the expression acts like an ALGOL by-value parameter in all situations.
  - If ALGOL calls a FORTRAN77 program unit and passes a by-name parameter to a by-value-result formal parameter, the parameter acts like a FORTRAN77 by-value-result parameter.
  - If ALGOL calls a FORTRAN77 program unit and passes a by-value parameter, the parameter acts like an ALGOL by-value parameter in all situations.

Example of Binding an ALGOL Subprogram Into a FORTRAN77 Host Program

The following example illustrates a FORTRAN77 host program, an ALGOL subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

FORTRAN77 Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE F77/HOST WITH FORTRAN77 LIBRARY;
      FORTRAN77 DATA
$ SET BINDINFO

C     EMPTY MAIN PROGRAM - IT WILL BE BOUND IN

      END

      SUBROUTINE WORK
      REAL A(3)
      DO 20 I = 1,4

C     THIS LOOP WILL CAUSE AN INVALID INDEX TO OCCUR

              A(I) = 1
20    CONTINUE
      END
? END JOB.
```

ALGOL Subprogram

Note that the ALGOL procedure contains an ON INVALIDINDEX statement that provides error recovery for the FORTRAN77 program.

```
? BEGIN JOB COMPILE/ALGOL;
      COMPILE ALGOL/SUBS ALGOL LIBRARY;
      ALGOL DATA
      [PROCEDURE WORK; EXTERNAL;]
      PROCEDURE MAIN;
        BEGIN
        LABEL XIT;
        FILE RMT (KIND=REMOTE);
        ON INVALIDINDEX:
           BEGIN
           WRITE (RMT,<" INVALID INDEX ">);
           GO TO XIT;
           END;
        WORK;
      XIT;
        END;
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/INVALID/INDEX;
      BIND PROG BINDER LIBRARY;
      BINDER DATA
      HOST IS F77/HOST;
      BIND .MAIN. FROM ALGOL/MAIN;
      USE .MAIN. FOR MAIN;
? END JOB.
```

The result of the bind is a file titled PROG. The execution of PROG generates the following output:

```
INVALID INDEX
```

Example of Replacing a FORTRAN77 Character Function by an ALGOL Procedure

The following example shows how a FORTRAN77 character function can be replaced by an ALGOL procedure with two leading parameters. The first parameter is an EBCDIC array with star bounds. The second parameter is an INTEGER variable that contains the length. The WFL job used to compile each program appears in bold type.

FORTRAN77 Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE F77/HOST WITH FORTRAN77 LIBRARY;
      FORTRAN77 DATA
$ SET BINDINFO
      EXTERNAL C
      CHARACTER*6  C, CL
      CL = C(2)
      PRINT *,CL
      END
? END JOB.
```

ALGOL Subprogram

```
? BEGIN JOB COMPILE/ALGOL;
      COMPILE ALGOL/SUBS ALGOL LIBRARY;
      ALGOL DATA
      PROCEDURE C (A,L,I);
      EBCDIC ARRAY A[*];
      INTEGER L, I;
         BEGIN
         REPLACE A[0] BY "2" FOR I, "4" FOR L-I;
         END;
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/CHARACTERS;
      BIND PROG BINDER LIBRARY;
      BINDER DATA
      HOST IS F77/HOST;
      BIND = FROM ALGOL/=;
? END JOB.
```

The result of the bind is a file titled PROG. The execution of PROG generates the following output:

```
224444
```

Example of Binding FORTRAN77 Program Units Into an ALGOL Host Program

During a CANDE session, the following two files are created and compiled:

File Name: ALGOL/HOST

```
$ SET WRITEAFTER
BEGIN
   FILE FILE6 (KIND=PRINTER);
   REAL ARRAY COMM [0;4];
   %
   % Array COMM is implicitly initialized when MAIN is bound in,
   % even though MAIN is not referenced as a subprogram.
   %
   PROCEDURE MAIN;  EXTERNAL;
   PROCEDURE F77SUB;  EXTERNAL;
   %
   WRITE (FILE6, */, COMM);
   F77SUB;
END.
```

File Name: F77/SEP

```
$ SET SEPARATE
      PROGRAM MAIN
         REAL C(5)
         COMMON /COMM/ C
         DATA C /1, 2, 3, 4, 5/
      END

      SUBROUTINE F77SUB
         REAL C(5)
         COMMON /COMM/ C
         WRITE (6, *) 'IN SUBROUTINE F77SUB, C = ', C
      END
```

The two code files are bound together by the following Binder program:

```
HOST IS OBJECT/ALGOL/HOST;
BIND MAIN, F77SUB FROM OBJECT/F77/SEP;
USE COMM FOR /COMM/;
```

The execution of the resulting code file produces the following printed output:

```
COMM[0]=1.0, COMM[1]=2.0, COMM[2]=3.0, COMM[3]=4.0, COMM[4]=5.0,
  IN SUBROUTINE F77SUB, C = 1.0  2.0  3.0  4.0  5.0
```

ALGOL-NEWP Interlanguage Binding

ALGOL-NEWP interlanguage binding is restricted to binding DCALGOL subprograms into a Master Control Program (MCP) host program compiled in NEWP. Binder cannot bind DCALGOL subprograms to other host programs compiled in NEWP.

Replacement binding is not allowed for procedures in the NEWP host program, except for externals that were bound in and must be rebound.

Observe the following requirements when binding a DCALGOL subprogram into an MCP host program compiled in NEWP:

- You must compile the subprogram in DCALGOL.
- You must declare the subprogram as external in the MCP code file.
- The subprogram cannot add globals to the MCP host program or contain OWN variable declarations.
- The only global variables that the subprogram can reference are those that are declared in the outer block of the MCP host program.

Note: *Because of the interaction with the NEWP SEPCOMP facility, the old object code of procedures being rebound is retained in the MCP code file. The old object code is not referenced and cannot be executed. If many rebindings occur, the MCP code file can grow undesirably large with the accumulation of useless object code. You can reallocate segments in the MCP by recompiling the MCP with the NEWP compiler.*

ALGOL-Pascal Interlanguage Binding

ALGOL-Pascal interlanguage binding consists of binding an ALGOL subprogram into a Pascal host program. Table 5–8 matches identifier types between ALGOL and Pascal.

Table 5–8. Corresponding Identifier Types between ALGOL and Pascal

ALGOL	Pascal
REAL ARRAY [*]	array of real array of record array of set array of vlstring array of packed array array of explicit type long set (> 48 elements in set) record vlstring explicit record (by-value) packed array of real packed array of set packed array of record packed array of vlstring
INTEGER ARRAY [*]	array of integer array of char array of enumeration array of fixed (n < 12) array of sfixed (n < 12) array of integer subrange array of char subrange array of enumeration subrange packed array of integer packed array of fixed (n < 12) packed array of sfixed (n < 12) packed array of subrange (> 256 elements in subrange) packed array of enumeration (> 256 elements in enumeration)

continued

Table 5–8. Corresponding Identifier Types between ALGOL and Pascal
(cont.)

ALGOL	Pascal
BOOLEAN ARRAY [*]	array of Boolean
DOUBLE ARRAY [*]	array of fixed (n > 11) array of sfixed (n > 11) packed array of fixed (n > 11) packed array of sfixed (n > 11)
HEX ARRAY [*]	hex (n) digits (n) s_digits (n) digits_s (n) Boolean1 Boolean4 packed array of Boolean packed array of subrange (0–16 elements in subrange) packed array of enumeration (0–16 elements in enumeration)
EBCDIC ARRAY [*]	bits (n) binary (n) u_display (n) z_display (n) display_z (n) s_display (n) display_s (n) word48 (n) word96 (n) integer48 integer96 real48 explicit record (var) packed array of char packed array of subrange • elements in subrange) packed array of enumeration (17–256 elements in enumeration)

continued

Table 5–8. Corresponding Identifier Types between ALGOL and Pascal
(cont.)

ALGOL	Pascal
REAL VARIABLE	real short set (1–48 elements in set)
INTEGER VARIABLE	integer char enumeration fixed (n < 12) sfixed (n < 12) integer subrange char subrange enumeration subrange
BOOLEAN VARIABLE	Boolean Boolean subrange
DOUBLE VARIABLE	fixed (n > 11) sfixed (n > 11)
PROCEDURE	procedure
REAL PROCEDURE	function : real
INTEGER PROCEDURE	function : integer function : char function : enumeration function : fixed (n < 12) function : sfixed (n < 12) function : integer subrange function : char subrange function : enumeration subrange
BOOLEAN PROCEDURE	function : Boolean function : Boolean subrange
DOUBLE PROCEDURE	function : fixed (n > 11) function : sfixed (n > 11)

Global Items

Pascal and ALGOL programs can share global items. When binding global items from an ALGOL subprogram into a Pascal host program, you must write a Pascal module heading that describes the ALGOL subprogram in Pascal terms. You include ALGOL global variables in the export declaration of the Pascal module heading as shown in the following portion of Pascal syntax:

```
MODULE m EXTERNAL;
   EXPORT int( a, p, f );
     VAR  a : integer;
     PROCEDURE p ( param : integer);
     FUNCTION  f : integer;
  END;
```

The EXTERNAL directive indicates that the module is written in a language other than Pascal. When a Pascal host program is compiled with modules that are declared with the EXTERNAL directive or modules that use other modules that are declared as external, the Pascal compiler creates a BINDERINPUT file. This file contains a set of suggested commands for Binder to use when binding the procedures compiled in the other language.

For example, when binding an ALGOL subprogram into a Pascal host program, the Pascal compiler puts USE statements in the BINDERINPUT to equate variable identifiers in Pascal and ALGOL. The USE statements are necessary because the Pascal compiler names the Pascal identifier by assigning the module name followed by a slash (/) and the ALGOL identifier name.

For example, assuming that the external module is titled m and an ALGOL variable is declared as a, as in the preceding example, the BINDERINPUT file would contain the following Binder USE statement:

```
USE M/A FOR A
```

There might be times when you need to edit the BINDERINPUT file. The internal name of the file for file equation is BINDERINPUT.

For more information about the BINDERINPUT file, the EXTERNAL directive, and modules, refer to the *Pascal Programming Reference Manual, Volume 1*.

Parameters

The following restrictions apply to parameters passed between ALGOL and Pascal:

- You cannot pass text files between ALGOL and Pascal.
- You can pass standard files between ALGOL and Pascal; however, you must declare in the Pascal host program the files that can be passed. Refer to the example following this discussion to see the code for a Pascal host program that passes standard files. For more information on Pascal file syntax, refer to the *Pascal Reference Manual, Volume 1*.

- Procedures and functions are allowed as parameters to ALGOL procedures bound to Pascal programs.
- Parameters passed between a Pascal host program and an ALGOL subprogram must match.
- Variables passed by reference (that is, variable parameters) in a Pascal host program must match by-name parameters in the ALGOL subprogram.
- Variables passed by value in a Pascal host program must match by-value parameters in the ALGOL subprogram.
- Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

Examples of Binding an ALGOL Subprogram Into a Pascal Host Program

Example 1

The following example shows how a Pascal program can incorporate a module written in ALGOL. The module heading describes an ALGOL procedure with one global variable, one untyped procedure, and one typed procedure to be bound into a Pascal program or library. The WFL job used to compile each program appears in bold type.

Pascal Host Program

```
? BEGIN JOB COMPILE/HOST;
  COMPILE PASCAL/HOST WITH PASCAL LIBRARY;
  PASCAL DATA CARD
  MODULE m EXTERNAL;
     EXPORT int ( a, p, f );
     VAR  a : integer;
     PROCEDURE p ( param : integer);
     FUNCTION  f : integer;
  END;
  PROGRAM prog;
     IMPORT int;
     VAR ig : integer;
  BEGIN
  p (42);
  ig := f;
  DISPLAY (concat ('value of a is ', string(a)));
  DISPLAY (concat ('value of ig is ', string(ig)));
  END.
? END JOB.
```

The BINDERINPUT file created by the Pascal compiler is as follows. You can use this file to bind the Pascal host program, PASCAL/HOST, and the ALGOL subprogram, OBJECT/M.

```
$ RESET LIST
 USE M/A FOR A;
 USE M/F FOR F;
 USE M/P FOR P;
 BIND
    M/F,
    M/P,
        DUMMY FROM OBJECT/M;
 HOST IS PASCAL/HOST;
```

ALGOL Subprogram

```
? BEGIN JOB MODULE/BODY;
  COMPILE OBJECT/M WITH ALGOL LIBRARY;
  ALGOL DATA CARD
$ SET LEVEL 3 LIBRARY
  [ INTEGER A; ]
  PROCEDURE P ( I );
```

```
      VALUE I; INTEGER I;
      BEGIN
      DISPLAY ("CALL ON P EXECUTED WITH I = " CAT STRING(I,*));
      A := 399;
      END;
  INTEGER PROCEDURE F;
      BEGIN
      DISPLAY ("CALL ON F EXECUTED");
      F := 7;
      END;
? END JOB.
```

When executed, the newly bound program displays the following:

```
CALL ON P EXECUTED WITH I = 42
CALL ON F EXECUTED
value of a is 399
value of ig is 7
```

Example 2

The following example shows a Pascal host program that has an ALGOL procedure bound into it. In this example, the formal parameter f represents an ALGOL file. In the Pascal host program, this formal parameter is compatible with any standard file parameter.

For this example, *FILE OF char* is the standard file parameter. Note that the Pascal buffer variable f@, is not affected by any input or output that occurs during the execution of the bound-in procedure.

```
MODULE m EXTERNAL;
   EXPORT i(p);
   PROCEDURE p (VAR f; stdfile )
END;

PROGRAM p;
   IMPORT i;
   TYPE tf= FILE OF char;
   VAR myf; tf;
BEGIN
   p( myf )
END.
```

COBOL-C Interlanguage Binding

COBOL-C interlanguage binding consists of binding a level-3 COBOL subprogram into a C host program. You declare a COBOL subprogram in C as a void function with COBOL linkage.

A COBOL subprogram cannot call functions with C linkage, and thus, it cannot call functions defined in the C host program. Identifiers that name data objects cannot be shared between C and COBOL, because a C program stores its external data items in its program heap.

If you are binding COBOL85 to C, ensure that the value of the FARHEAP option matches. If the FARHEAP option is false, the value of the MEMORY_MODEL must match the C host. Table 5–9 shows the corresponding parameter types between C and COBOL.

Table 5–9. Corresponding Parameter Types Between C and COBOL

C Argument Type	COBOL TYPE
char	77 BINARY BY CONTENT
int, short, long	77 BINARY BY CONTENT
pointer	77 BINARY BY CONTENT
float	77 REAL BY CONTENT
long double	77 DOUBLE BY CONTENT
int&, short&, long&	77 BINARY BY REFERENCE
pointer&	77 BINARY BY REFERENCE
float&	77 FLOAT BY REFERENCE
long double&	77 DOUBLE BY REFERENCE

Example of COBOL-C Binding

The following example illustrates binding a COBOL subprogram into a C host. The WFL jobs used to compile and bind appear in bold type.

C Host Program

```
? BEGIN JOB C/HOST;
  COMPILE OBJECT/C/HOST CC LIBRARY;
  CC DATA
       extern "COBOL" void COBOLSUBPROGRAM (int, int(&));
       main ()
         {int passedByValue = 1, passedByReference = 1;
         COBOLSUBPROGRAM (passedByValue, passedByReference);
         COBOLSUBPROGRAM (passedByValue, passedByReference);
         return 0;}
? END JOB.
```

COBOL Subprogram

```
? BEGIN JOB COMPILE/COBOL;
  COMPILE OBJECT/COBOL/SUBPROGRAM COBOL74 LIBRARY;
  COBOL DATA
      $$ SET LEVEL = 3
       IDENTIFICATION DIVISION.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       77 C-INT-VALUE        PIC S9(11) BINARY BY CONTENT.
       77 C-INT-REFERENCE    PIC S9(11) BINARY BY REFERENCE.
       PROCEDURE DIVISION USING C-INT-VALUE C-INT-REFERENCE.
       SUBPROGRAM SECTION.
       DISPLAY-C-ARGUMENTS.
           DISPLAY "C-INT-VALUE IS " C-INT-VALUE
                   " AND C-INT-REFERENCE IS " C-INT-REFERENCE.
           ADD 1 TO C-INT-VALUE C-INT-REFERENCE.
       EXIT-PROCEDURE.
           EXIT PROCEDURE.
? END JOB.
```

Binder Input File

```
? BEGIN JOB COBOL/C/BIND;
  BIND OBJECT/COBOL/C/BIND BINDER;
  BINDER DATA
    BIND ? FROM OBJECT/C/HOST;
    BIND COBOLSUBPROGRAM FROM OBJECT/COBOL/SUBPROGRAM;
? END JOB.
```

The result of the bind is a program titled OBJECT/COBOL/C/BIND, which displays the following information:

```
C-INT-VALUE IS +00000000001 AND C-INT-REFERENCE IS +00000000001.
C-INT-VALUE IS +00000000001 AND C-INT-REFERENCE IS +00000000002.
```

COBOL-FORTRAN Interlanguage Binding

COBOL-FORTRAN interlanguage binding consists of binding a COBOL program into a FORTRAN host program or binding a FORTRAN subprogram into a COBOL host program. Table 5–10 matches identifier types between COBOL and FORTRAN.

Table 5–10. Corresponding Identifier Types between COBOL and FORTRAN

COBOL	COBOL74	FORTRAN
COMP OCCURS	REAL OCCURS	REAL ARRAY/COMMON BLOCK
COMP OCCURS	BINARY OCCURS	INTEGER ARRAY/COMMON BLOCK
COMP OCCURS	BINARY OCCURS	LOGICAL ARRAY/COMMON BLOCK
	DOUBLE PRECISION ARRAY/COMMON BLOCK	
	COMPLEX ARRAY/COMMON BLOCK	
COMP-2		
ASCII		
DISPLAY	DISPLAY	
DISPLAY LOWERBOUNDS + LEVEL 77	DISPLAY LOWERBOUNDS + LEVEL 77	
COMP-4	REAL	REAL VARIABLE
COMP or COMP-1	BINARY (1 to 11 digits)	INTEGER VARIABLE
COMP or COMP-1	BINARY (1 to 11 digits)	LOGICAL VARIABLE
COMP-5	DOUBLE	DOUBLE PRECISION VARIABLE
	COMPLEX VARIABLE	
SECTION	SECTION	SUBROUTINE
	FUNCTION	
SECTION + 2 PARAMETERS (DISPLAY LOWERBOUNDS + LEVEL 77)	SECTION + 2 PARAMETERS (DISPLAY LOWERBOUNDS + LEVEL 77)	
FILE	FILE	FILE
DIRECT FILE	DIRECT FILE	

Global Items

Only files and FORTRAN subprograms can be shared globally between COBOL and FORTRAN. Files in FORTRAN are given the internal name, FILEnn, where nn is a 1-digit or 2-digit number (without a leading zero) that refers to the unit number in a FORTRAN I/O statement.

For example, a WRITE(6,1) statement in a FORTRAN subroutine writes to FILE6. To share a common file with FORTRAN, a COBOL file must be named and declared accordingly.

Parameters

The following restrictions apply when passing parameters between COBOL and FORTRAN:

- You can pass only arrays and simple variables (declared as 77-level items in COBOL) as parameters between FORTRAN and COBOL.
- You must declare COBOL array parameters as REAL or BINARY in COBOL74 and COBOL85, or COMPUTATIONAL in COBOL68, because FORTRAN works only with word-oriented arrays.
- When passing an array from FORTRAN to COBOL, include a LOWER-BOUNDS clause in the 01-level description for the array. When passing an array from COBOL to FORTRAN, you must also include a LOWER-BOUNDS clause in the LOCAL-STORAGE SECTION description of the formal parameters.
- COBOL always assumes that the lower bound of an array that is passed or received is 0 (zero).
- Unpredictable results can occur if you pass to COBOL a FORTRAN subscripted variable with a value other than 0 (zero).
- You should pass only the first array appearing in a FORTRAN common block to COBOL. Otherwise, results are unpredictable.
- You cannot pass subroutines and functions as parameters between COBOL and FORTRAN.
- Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

COBOL-FORTRAN77 Interlanguage Binding

COBOL-FORTRAN77 interlanguage binding consists of binding either a COBOL program into a FORTRAN77 host program or a FORTRAN77 subprogram into a COBOL host program. Table 5–11 matches identifier types between COBOL and FORTRAN77.

Table 5–11. Corresponding Identifier Types between COBOL and FORTRAN77

COBOL	COBOL74	FORTRAN77
COMP OCCURS	REAL OCCURS	REAL ARRAY/COMMON BLOCK
COMP OCCURS	BINARY OCCURS	INTEGER ARRAY/COMMON BLOCK
COMP OCCURS	BINARY OCCURS	LOGICAL ARRAY/COMMON BLOCK
COMP-2		
ASCII		
DISPLAY	DISPLAY	CHARACTER COMMON BLOCK
DISPLAY LOWERBOUNDS + LEVEL 77	DISPLAY LOWERBOUNDS + LEVEL 77	CHARACTER ARRAY/CHARACTER VARIABLE
		DOUBLE PRECISION ARRAY
		COMPLEX ARRAY
COMP-4	REAL	REAL VARIABLE
COMP or COMP-1	BINARY (1 to 11 digits)	INTEGER VARIABLE
COMP or COMP-1	BINARY (1 to 11 digits)	LOGICAL VARIABLE
COMP-5	DOUBLE	DOUBLE PRECISION VARIABLE
		COMPLEX VARIABLE
SECTION	SECTION	SUBROUTINE/MAIN PROGRAM
		FUNCTION
SECTION + 2 PARAMETERS (DISPLAY LOWERBOUNDS + LEVEL 77)	SECTION + 2 PARAMETERS (DISPLAY LOWERBOUNDS + LEVEL 77)	CHARACTER FUNCTION
FILE	FILE	FILE
DIRECT FILE	DIRECT FILE	

Global Items

Only files and FORTRAN77 subprograms can be shared globally between COBOL and FORTRAN77. Files in FORTRAN77 are given the internal name, FILEnn, where nn is a 1-digit or 2-digit number (without a leading zero) that refers to the unit number in a FORTRAN77 I/O statement. For example, a WRITE(6,1) statement in a FORTRAN77 subroutine writes to FILE6. To share a common file with FORTRAN77, a COBOL file must be named and declared accordingly.

You can simulate a character function in COBOL with a COBOL section that has two leading parameters. These parameters must be a DISPLAY array with the LOWER-BOUNDS clause and a 77-level item declared BINARY in COBOL74 and COBOL85, or COMP-1 in COBOL68. The second parameter corresponds to the character length of the character function.

Parameters

The following restrictions apply to parameters passed between FORTRAN77 and COBOL:

- You can pass only simple characters, arrays, and variables (declared as 77-level items in COBOL) as parameters between FORTRAN77 and COBOL.
- When passing an array from FORTRAN77 to COBOL, include a LOWER-BOUNDS clause in the 01-level description for the array. When passing an array from COBOL to FORTRAN77, you must also include a LOWER-BOUNDS clause in the LOCAL-STORAGE SECTION description of the formal parameters.
- COBOL always assumes that the lower bound of an array that is passed or received is 0 (zero).
- Unpredictable results can occur if you pass to COBOL a FORTRAN77 subscripted variable with a value other than 0 (zero).
- You should pass only the first array appearing in a FORTRAN77 common block to COBOL. Otherwise, the results are unpredictable.
- All FORTRAN77 character variables are stored in a character pool. You should pass to COBOL only the first FORTRAN77 character variable in the pool. Otherwise, the results are unpredictable.
- You cannot pass subroutines and functions as parameters between COBOL and FORTRAN77.
- Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

Example of Passing a FORTRAN77 Character Variable to a COBOL74 Section

The following example shows a FORTRAN77 host program, a COBOL subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

FORTRAN77 Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE F77/HOST WITH FORTRAN77 LIBRARY;
      FORTRAN77 DATA
$ SET BINDINFO
      CHARACTER*7 C
      CALL SUB (C)
      PRINT *,C
      END
? END JOB.
```

COBOL Subprogram

```
? BEGIN JOB COMPILE/COBOL74;
      COMPILE COBOL74/SUB COBOL74 LIBRARY;
      COBOL74 DATA
      $ SET LEVEL = 3
        IDENTIFICATION DIVISION.
        PROGRAM-ID. EXAMPLE.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. A-15.
        OBJECT-COMPUTER. A-15.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01 COBARY DISPLAY LOWER-BOUNDS RECEIVED BY REFERENCE.
           03 DUMMY    PIC X(1) OCCURS 7 TIMES.
        01 FAKEIT REDEFINES COBARY.
           03 NUMB     PIC X(7).
        77 LEN BINARY  PIC 9(11).
        PROCEDURE DIVISION USING COBARY, LEN.
        CB SECTION.
        STORE-VALUE.
           MOVE "ABCDEFG" TO NUMB.
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/COBOL74;
      BIND PROG BINDER LIBRARY;
      BINDER DATA
      HOST IS F77/HOST;
      BIND SUB FROM COBOL74/SUB;
      USE SUB FOR CB;
? END JOB.
```

The result of the bind is an object file titled PROG. When executed, PROG generates the following output:

```
ABCDEFG
```

COBOL-Pascal Interlanguage Binding

COBOL-Pascal interlanguage binding consists of binding a COBOL subprogram into a Pascal host program. Table 5–12 matches identifier types between COBOL and Pascal.

Table 5–12. Corresponding Identifier Types between COBOL and Pascal

COBOL	COBOL74	Pascal
COMP OCCURS	REAL OCCURS	array of real
		array of record
		array of set
		array of vlstring
		array of packed array
		array of explicit type
		long set (> 48 elements in set)
		record
		vlstring
		explicit record (by-value)
		packed array of real
		packed array of set
		packed array of record
		packed array of vlstring

continued

Table 5–12. Corresponding Identifier Types between COBOL and Pascal (cont.)

COBOL	COBOL74	Pascal
COMP OCCURS	BINARY OCCURS	array of integer array of char array of enumeration array of fixed (n < 12) array of sfixed (n < 12) array of integer subrange array of char subrange array of enumeration subrange packed array of integer packed array of fixed (n < 12) packed array of sfixed (n < 12) packed array of subrange (> 256 elements in subrange) packed array of enumeration (> 256 elements in enumeration)
COMP OCCURS	BINARY OCCURS	array of Boolean array of fixed (n > 11) array of sfixed (n > 11) packed array of fixed (n > 11) packed array of sfixed (n > 11) array of fixed (n > 11)array of sfixed (n > 11) packed array of fixed (n > 11) packed array of sfixed (n > 11)

continued

Table 5–12. Corresponding Identifier Types between COBOL and Pascal (cont.)

COBOL	COBOL74	Pascal
COMP-2		hex (n)
		digits (n)
		s_digits (n)
		digits_s (n)
		Boolean1
		Boolean4
		packed array of Boolean
		packed array of subrange
		(0-16 elements in subrange)
		packed array of enumeration
		(0-16 elements in enumeration)
DISPLAY	DISPLAY	bits (n)
		binary (n)
		u_display (n)
		z_display (n)
		display_z (n)
		s_display (n)
		display_s (n)
		word48 (n)
		word96 (n)
		integer48
		integer96
		real48
		explicit record (var)
		packed array of char
		packed array of subrange
		(17-256 elements in subrange)
		packed array of enumeration
		(17-256 elements in enumeration)

continued

Table 5–12. Corresponding Identifier Types between COBOL and Pascal (cont.)

COBOL	COBOL74	Pascal
COMP-4	REAL	real
		short set
		(1-48 elements in set)
COMP OR COMP-1	BINARY (1 to 11 digits)	integer
		char
		enumeration
		fixed (n < 12)
		sfixed (n < 12)
		integer subrange
		char subrange
		enumeration subrange
COMP OR COMP-1	BINARY (1 to 11 digits)	Boolean
		Boolean subrange
COMP-5	DOUBLE	fixed (n > 11)
		sfixed (n > 11)
SECTION	SECTION	procedure
		function : real
		function : integer
		function : char
		function : enumeration
		function : fixed (n < 12)
		function : sfixed (n < 12)
		function : integer subrange
		function : char subrange
		function : enumeration subrange
		function : Boolean
		function : Boolean subrange
		function : fixed (n > 11)
		function : sfixed (n > 11)

Global Items

You can share global items between Pascal and COBOL. If a COBOL subprogram is to reference a global variable in a Pascal host program, you must declare the variable by using the GLOBAL clause or the GLOBAL compiler control option in the COBOL subprogram.

When binding global items from a COBOL subprogram into a Pascal host program, you must write a Pascal module heading that describes the COBOL subprogram in Pascal terms. You include COBOL global variables in the export declaration of the Pascal module heading as shown in the following portion of Pascal syntax:

```
MODULE m EXTERNAL;
   EXPORT int( a, p);
     VAR  a : integer;
     PROCEDURE p (var param : integer);
  END;
```

The EXTERNAL directive indicates that the module is written in a language other than Pascal. When a Pascal host program is compiled with modules that are declared with the EXTERNAL directive or modules that use other modules that are declared as EXTERNAL, the Pascal compiler creates a BINDERINPUT file. This file contains a set of suggested commands for Binder to use when binding the procedures compiled in the other language.

For example, when binding a COBOL subprogram into a Pascal host program, the Pascal compiler puts USE statements in the BINDERINPUT to equate variable identifiers in Pascal and COBOL. The USE statements are necessary because the Pascal compiler names the Pascal identifier by assigning the module name followed by a slash (/) and the COBOL identifier name.

For example, assuming that the external module is named m and the COBOL variables are declared as a and p, as in the preceding example, the BINDERINPUT file would contain the following Binder USE statement:

```
USE M/A FOR A;
USE M/P FOR P;
```

There might be times when you need to edit the BINDERINPUT file. The internal name of the file for file equation is BINDERINPUT.

For more information about the BINDERINPUT file, the EXTERNAL directive, and modules, refer to the *Pascal Reference Manual*, *Volume 1*.

Parameters

The following restrictions apply when passing parameters between Pascal and COBOL:

- When passing a word-oriented variable or array (integer, real, or Boolean) between Pascal and COBOL68, declare the word-oriented entity as COMPUTATIONAL in the COBOL68 program. You can declare real variables as COMPUTATIONAL-4.

- In a COBOL74 program, you must declare a real array or variable as REAL, and an integer or Boolean array or variable as BINARY.
- You cannot pass text files between Pascal and COBOL.
- You can pass standard files between COBOL and Pascal; however, you must declare in the Pascal host program the files that can be passed. Refer to the example following this discussion to see the code for a Pascal host program that passes standard files. For more information on Pascal file syntax, refer to the *Pascal Reference Manual*, *Volume 1*.
- Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

Example of Binding a COBOL74 Procedure Into a Pascal Host Program

The following example shows how a Pascal program can incorporate a module written in another language. The module heading describes a COBOL74 procedure with one global variable to be bound into a Pascal program or module. The WFL job used to compile each program appears in bold type.

Pascal Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE PASCAL/HOST WITH PASCAL LIBRARY;
      PASCAL DATA CARD
      MODULE m EXTERNAL;
         EXPORT int( a, p);
         VAR  a : integer;
         PROCEDURE p(var param : integer);;
      END;
      PROGRAM prog;
         IMPORT int;
         VAR i : integer;
      BEGIN
      p(i);
      DISPLAY (concat ('value of i is ',string(i)));
      DISPLAY (concat ('value of a is ',string(a)));
      END.
 ? END JOB.
```

The Pascal compiler produces the following BINDERINPUT file. You can use this file to bind the Pascal host program, PASCAL/HOST, and the COBOL subprogram, OBJECT/M.

```
$ RESET LIST
USE M/A FOR A;
USE M/P FOR P;
BIND
   M/P,
       DUMMY FROM OBJECT/M;
HOST IS PASCAL/HOST;
```

COBOL74 Subprogram

```
? BEGIN JOB MODULE/BODY;
      COMPILE OBJECT/M WITH COBOL74 LIBRARY;
      COBOL74 DATA CARD
      $ SET LEVEL = 3
        IDENTIFICATION DIVISION.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        77 A   PIC S9(11) GLOBAL BINARY.
        77 I   PIC S9(11) LOCAL BINARY RECEIVED BY REFERENCE.
        PROCEDURE DIVISION USING I.
        LBL.
           DISPLAY "CALL ON SUBPROGRAM EXECUTED".
           MOVE 111 TO I.
           MOVE 399 TO A.
           EXIT PROCEDURE.
? END JOB.
```

When executed, the bound program generates the following output:

```
CALL ON SUBPROGRAM EXECUTED
value of i is 111
value of a is 399
```

Example of Binding a COBOL Procedure Into a Pascal Host Program

The following example shows a Pascal host program that has a procedure bound into it. In this example, the formal parameter (f) represents a COBOL file. In the Pascal host program, this formal parameter is compatible with any standard file parameter. For this example, *FILE OF char* is the standard file parameter. Note that the Pascal buffer variable *f@* is not affected by any input or output that occurs during the execution of the bound-in procedure.

```
MODULE m EXTERNAL;
   EXPORT i(p);
   PROCEDURE p (VAR f: stdfile )
END;
PROGRAM p;
   IMPORT i;
   TYPE tf= FILE OF char;
   VAR myf: tf;
BEGIN
   p( myf )
END.
```

FORTRAN-FORTRAN77 Interlanguage Binding

FORTRAN-FORTRAN77 interlanguage binding consists of binding a FORTRAN subprogram into a FORTRAN77 host program or binding a FORTRAN77 subprogram into a FORTRAN host program. You cannot bind a FORTRAN77 subroutine with a label parameter into a FORTRAN host program.

Table 5–13 matches identifier types between FORTRAN and FORTRAN77.

Table 5–13. Corresponding Identifier Types between FORTRAN and FORTRAN77

FORTRAN	FORTRAN77
REAL ARRAY/COMMON BLOCK	REAL ARRAY/COMMON BLOCK
INTEGER ARRAY/COMMON BLOCK	INTEGER ARRAY/COMMON BLOCK
LOGICAL ARRAY/COMMON BLOCK	LOGICAL ARRAY/COMMON BLOCK
DOUBLE PRECISION ARRAY/COMMON BLOCK	COMMON BLOCK
COMPLEX ARRAY/COMMON BLOCK	COMMON BLOCK
	CHARACTER COMMON BLOCK
	CHARACTER ARRAY/CHARACTER VARIABLE
	DOUBLE PRECISION ARRAY
	COMPLEX ARRAY
REAL VARIABLE	REAL VARIABLE
INTEGER VARIABLE	INTEGER VARIABLE
LOGICAL VARIABLE	LOGICAL VARIABLE
DOUBLE PRECISION VARIABLE	DOUBLE PRECISION VARIABLE
COMPLEX VARIABLE	COMPLEX VARIABLE
SUBROUTINE	SUBROUTINE/MAIN PROGRAM
FUNCTION	FUNCTION
	CHARACTER FUNCTION
FILE	FILE

Subprograms

A FORTRAN subprogram can be a FORTRAN subroutine or function. A FORTRAN77 subprogram can be a FORTRAN77 main program, subroutine, function, or block data subprogram.

A FORTRAN77 main program is compatible with a FORTRAN subroutine that has no parameters. Thus, you can bind a FORTRAN77 main program into a FORTRAN77 host program by replacing the main program with a separately compiled FORTRAN subroutine.

Use the following Binder syntax to indicate the title of the file containing the FORTRAN subroutine and to indicate the name of the FORTRAN subroutine to use in place of the FORTRAN77 main program, *.MAIN.:*

```
BIND .MAIN. FROM <file specifier>
USE .MAIN. FOR <identifier>
```

Unlike FORTRAN77 main programs, FORTRAN main programs cannot be bound or replacement bound by a host program.

Exported subroutines and functions can be replacement bound. It is not possible to add new exported program units to a host program.

Common Blocks

FORTRAN77 arithmetic common blocks correspond to FORTRAN common blocks. However, FORTRAN77 accesses the common block only through a single-precision descriptor and is not affected by odd offsets.

When a common block is bound, its resulting length is the longest of all the lengths declared for that block in the host program and bound subprograms, unless the FORTRAN77 compiler control option CODEFILEINIT is set in the host program.

Any common block that has been code file initialized cannot be extended.

Parameters

FORTRAN77 double-precision and complex arrays are passed to subprograms as single-precision descriptors. The array elements do not have to be on even-word boundaries. For this reason, the FORTRAN77 arrays do not correspond to any FORTRAN array and, thus, cannot be passed as parameters between the two languages. (In some cases, you can override this restriction by using the DOUBLEARRAYS compiler control option described in the *FORTRAN77 Reference Manual*.)

You cannot pass subroutines and functions as parameters between FORTRAN77 and FORTRAN.

Binder allows a procedure with unknown parameters to match and bind with a procedure of the same name with either known or unknown parameters.

Characters

FORTRAN77 character variables, character arrays, and character common blocks do not correspond to any FORTRAN data structure.

Libraries

You can bind or replacement bind subroutines and functions into a host program that references libraries. You can also add libraries to the host program.

When compiling subprograms, declare all libraries used by the subprograms before the first executable program unit.

Libraries in subprograms to be bound to a host program do not have to be explicitly declared in the host program. If libraries are not declared in the host program, Binder builds a library template from the binding information in the subprogram file. Once the template is built, Binder can add library objects not explicitly declared in the host.

Subprograms that do not reference libraries can be bound into host programs that reference libraries or that are themselves libraries.

FORTRAN describes all simple variable arguments to imported subprograms as call-by-reference. FORTRAN77 describes them as call-by-name. When calling a library object, Binder allows call-by-reference and call-by-name arguments to match at run time.

Example of Binding a FORTRAN Common Block Into a FORTRAN77 Host Program

The following example shows a FORTRAN77 host program, a FORTRAN subprogram, and the Binder input file used to bind them together. The WFL job used to compile each program appears in bold type.

FORTRAN77 Host Program

```
? BEGIN JOB COMPILE/HOST;
      COMPILE F77/HOST WITH FORTRAN77 LIBRARY;
      FORTRAN77 DATA
$ SET BINDINFO
      COMMON A,B,C,D
      DATA A,B,C,D /1,1,1,1/
      CALL SUB
      WRITE (6,*) A,B,C,D
      END
? END JOB.
```

FORTRAN Subprogram

```
? BEGIN JOB COMPILE/FORTRAN;
      COMPILE FORTRAN/SUB FORTRAN LIBRARY;
      FORTRAN DATA
$ SET SEPARATE
      SUBROUTINE SUB
      COMMON ONE, TWO
      DOUBLE PRECISION ONE, TWO
      TWO = 2
      END
? END JOB.
```

Binder Input File

```
? BEGIN JOB BIND/CHARACTERS;
      BIND PROG BINDER LIBRARY;
      BINDER DATA
      HOST IS F77/HOST;
      BIND = FROM FORTRAN/=;
? END JOB.
```

The result of the bind is an object file titled PROG. When executed, PROG generates the following output:

```
2*1.0  2.0  0.0
```

Example of Interlanguage Binding Involving FORTRAN77, COBOL74, and ALGOL

The following is a complex example of interlanguage binding. The host program is a FORTRAN77 program that passes an array as a parameter to a COBOL74 program. The COBOL74 program calls an ALGOL procedure, that in turn calls another COBOL74 program. The WFL job used to compile each program appears in bold type.

FORTRAN77 Host Program

The WFL job compiles and saves the program.

```
? BEGIN JOB COMPILE/HOST;
      COMPILE FORTRAN77/HOST FORTRAN LIBRARY;
      FORTRAN77 DATA
      DIMENSION A(7)

C   PLACE ALPHABET IN A(1)-A(5)
      CALL MOVE (A(1),"ABCDEFGHIJKLMNOPQRSTUVWXYZ    ",30)

C   NOW CALL THE COBOL PROGRAM
      CALL COBPRO(A)
      STOP
      END
? END JOB.
```

COBOL74 Subprogram

The following WFL job compiles the COBOL74 program called from the FORTRAN77 host program and saves it in the file named SEP/COBPRO.

```
? BEGIN JOB COMPILE/SEP/COBPRO;
      COMPILE SEP/COBPRO COBOL74 LIBRARY;
      COBOL74 DATA
      $ SET LEVEL = 3
        IDENTIFICATION DIVISION.
        PROGRAM-ID.  NUMBERS.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER.  A-15.
        OBJECT-COMPUTER.  A-15.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01 COBARY COMP LOWER-BOUNDS REFERENCE.
           03 DUMMY         PIC 9(11) OCCURS 7 TIMES.
        01 FAKEOUT REDEFINES COBARY.
           03 FILLER        PIC X(30).
           03 NUMB          PIC X(12).
        LOCAL-STORAGE SECTION.
        LD PASS.
        01 LARY COMP.
           03 OTHER-DUMMY   PIC 9(11) OCCURS 7 TIMES.
```

```
        PROCEDURE DIVISION USING COBARY.
        DECLARATIVES.
        A1 SECTION.
           USE EXTERNAL PROCEDURE WITH PASS USING LARY.
        END DECLARATIVES.
        S1 SECTION.
        PUT-IN-NUMBERS.
           MOVE "0123456789  "  TO NUMB.
           ENTER A1 USING COBARY.
? END JOB.
```

ALGOL Subprogram

The following WFL job compiles the ALGOL procedure called from the COBOL program and saves it in the file named SEP/ALG.

```
? BEGIN JOB COMPILE/SEP/ALG;
      COMPILE SEP/ALG ALGOL LIBRARY;
      ALGOL DATA
      [PROCEDURE COBPRINT(A,B); ARRAY A,B[0]; EXTERNAL;]
$ SET LEVEL 4
      PROCEDURE ALG (ARGOLD);
      ARRAY ARGOLD[0];
      BEGIN
        INTEGER M;
        ARRAY ARGNU [0;6];
        POINTER PN,PO,POT;
        PO := POT := POINTER(ARGOLD[6])+5;
        PN := POINTER(ARGNU);
        FOR M := 0 STEP 1 UNTIL 41 DO
           BEGIN
             PO := POT-M;
             REPLACE PN+M BY PO FOR 1;
           END;
        COBPRINT(ARGOLD,ARGNU);
      END;
? END JOB.
```

COBOL74 Subprogram

The following WFL job compiles the COBOL74 program called from the ALGOL procedure and saves it in the file named SEP/COBPRINT.

```
? BEGIN JOB COMPILE/SEP/COBPRINT;
      COMPILE SEP/COBPRINT COBOL74 LIBRARY;
      COBOL74 DATA
      $ SET LEVEL = 3
        IDENTIFICATION DIVISION.
        PROGRAM-ID.  PRINT/ARRAYS.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. A-15.
        OBJECT-COMPUTER. A-15.
```

```
          INPUT-OUTPUT SECTION.
          FILE-CONTROL.
             SELECT PR ASSIGN TO PRINTER.
          DATA DIVISION.
          FILE SECTION.
          FD PR.
          01 PR-RCD           PIC X(42).
          WORKING-STORAGE SECTION.
          01 A COMP REFERENCE.
             03 DUMMY         PIC 9(11) OCCURS 7 TIMES.
          01 B COMP REFERENCE.
             03 OTHER-DUMMY   PIC 9(11) OCCURS 7 TIMES.
          PROCEDURE DIVISION USING A B.
          CB SECTION.
          OPEN-PR.
             OPEN OUTPUT PR.
             WRITE PR-RCD FROM A.
             WRITE PR-RCD FROM B.
? END JOB.
```

Binder Input File

The four files are then bound and executed by the following WFL job:

```
? BEGIN JOB BIND/EXAMPLE/PROG;
      BIND EXAMPLE/PROG BINDER;
      BINDER DATA
      HOST IS FORTRAN77/HOST;;
      USE A1 FOR ALG;
      BIND A1 FROM SEP/ALG;
      BIND = FROM SEP/=;
      STOP;
? END JOB.
```

The result of the bind is an object file named EXAMPLE/PROG. When executed, EXAMPLE/PROG generates the following output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ    0123456789
  9876543210    ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Section 6
Binding Intrinsics

This section provides the information you need to compile, bind, and access an intrinsic file. For additional information about intrinsics, refer to the appropriate language manual.

What Is an Intrinsic?

An intrinsic is a program routine that performs common mathematical and other operations. An intrinsic file consists of standard system intrinsics such as SIN, SQRT, and formatting routines, as well as user-written intrinsics commonly referred to as *installation* intrinsics.

Although intrinsics can be written only in ALGOL, COBOL, and FORTRAN, almost any language that defines binding can access an intrinsic file. All compilers automatically recognize and access standard system intrinsics. COBOL and PL/I programs can automatically access installation intrinsics as well. FORTRAN and ALGOL programs must be compiled with the INSTALLATION compiler control option set in order to access installation intrinsics.

Compiling Intrinsics

When compiling an intrinsic, observe the following requirements:

- You must set the INTRINSICS compiler control option for all compilations.
- When compiling ALGOL and FORTRAN programs that access installation intrinsics, you must set the INSTALLATION compiler control option.
- When compiling an intrinsic in COBOL, you cannot reference global items.
- When compiling an intrinsic in DCALGOL, you can only reference other intrinsics or Master Control Program (MCP) items.

Creating a Binder Input File

To bind an intrinsic, you must create a Binder input file that includes the following:

- A *$SET INTRINSICS* Binder control record.
- A BIND statement specifying the source file or files for standard system intrinsics. Using the *BIND,=* form of the BIND statement causes Binder to look for all the standard system intrinsics whose names and intrinsic numbers are tabulated within Binder.
- One or more BIND statements that specify the installation intrinsics.

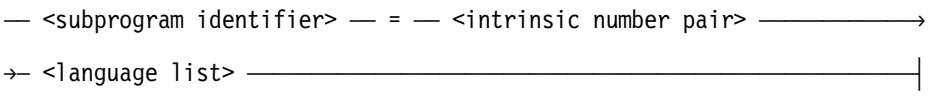
Once an intrinsic is bound into an intrinsic file, you cannot alter the intrinsic number, type of subprogram, or parameters by performing replacement binding. If you need to modify any of these items in an installation intrinsic, you must specify the necessary changes, and then bind the intrinsic into a new intrinsic file. To modify a standard system intrinsic, you must update the Binder internal tables and create a new intrinsic file.

Intrinsic Specification

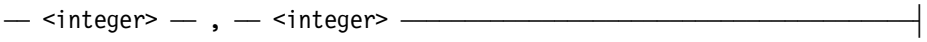
Use the intrinsic specification construct with the BIND statement to bind installation intrinsics.

Syntax

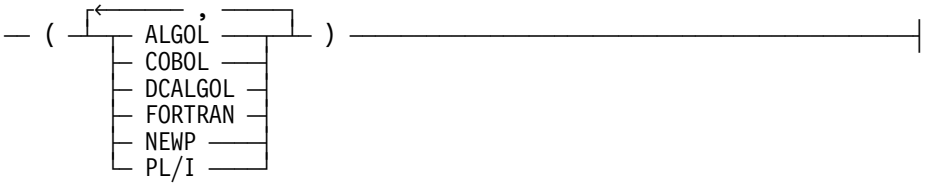
<intrinsic specification>



<intrinsic number pair>



<language list>



Explanation

<intrinsic number pair>

Specifies an installation number and an intrinsic number. The first integer of the intrinsic number pair metatoken specifies an installation number, which can range in value from 0 through 2046; however, numbers 0 through 99 are reserved for system use.

The second integer specifies an intrinsic number, which can range in value from 0 through 8191.

No two intrinsics within an intrinsic file can have the same intrinsic number pair.

<language list>

Specifies the compilers that are authorized to reference a given intrinsic. A referencing language is not necessarily the same language in which the intrinsic is written. For example, the DCALGOL language identifier allows a specified intrinsic to be accessed by the DMALGOL and DCALGOL compilers.

Details

Binder automatically binds standard system intrinsics that are referenced as EXTERNAL in a program. Thus, you do not need to specify such system intrinsics in a BIND statement.

Example

This example shows a Binder input file that is used to bind intrinsics.

```
$ SET INTRINSICS
      BIND = FROM INTR/=;
      BIND MYSIN = 101, 1 (ALGOL,FORTRAN) FROM INTL/=;
      BIND COFFEE = 102, 2 (COBOL) FROM POT;
      STOP;
```

Section 7
Binding Programs That Access Databases

You can bind programs that access Data Management System II (DMSII) or Semantic Information Manager (SIM) Databases. To do so, you must declare the database in the host program and meet the criteria discussed in this section.

Note that the examples in this section illustrate the possible combinations in which the host program and the subprogram can declare a SIM database for binding. These examples are not complete and cannot be compiled as shown. Comments are placed within the examples to indicate portions of missing code.

References made to *compiler* in the examples in this section refer to the language compiler used to compile the host program and the subprogram.

Binding DMSII Databases

You can bind subprograms that access DMSII databases to host programs compiled with ALGOL, COBOL85, COBOL74, COBOL68, and PL/I compilers. Observe the following requirements:

- The database code files must be compiled with a Mark 3.5 or later compiler.
- You must declare the DMSII database in the host program.
- The host program must invoke all the database structures that it references, as well as all the database structures referenced in the subprograms it is attempting to bind.
- You must declare the DMSII database as global in all subprograms that access the database.
- A database invocation in a subprogram must include the same structures in the same order as the host database invocation, regardless of whether the subprogram accesses a particular structure.
- You must compile the host program and all subprograms with the same DMSII description file. If making a DASDL update or reorganization requires you to recompile a component that accesses the database, then you must recompile all of the components that access the database, even if a particular component accesses an unchanged portion of the database.

Binding SIM Databases

You can bind subprograms that access SIM databases to host programs compiled in ALGOL, COBOL74, and Pascal. (Pascal programs can serve only as host programs.) You can bind subprograms that reference the following elements:

- A database declared in the host program
- An entity reference variable declared in the host program
- A query variable declared in the host program

Binder performs type checking of the variables for compatibility.

For an explanation of SIM concepts and instructions for using SIM, refer to the *InfoExec Administration Guide* and the *InfoExec Semantic Information Manager (SIM) Programming Guide*.

SIM Data Types

Binder recognizes three data types when binding programs that use SIM databases: the DMRECORD variable, the entity reference variable, and the query variable. Before binding programs, Binder verifies that these data types reference the same class in the same database. The three data types are as follows:

- DMRECORD

  A DMRECORD is made up of fields that hold information retrieved from SIM. You can bind DMRECORDs to each other.

- Entity reference variable

  An entity reference variable refers to an entity with the attributes of a given database class.

  The compiler queries SIM about this database class and gets information about the format of the entity reference variable. The format determines the number of words that are allocated for this variable. If a subprogram references an entity reference variable, the variable must be declared in the global declarations and must be preceded by the database declaration.

- Query variable

  A query variable represents an active query and contains information about the state of a query.

The compiler queries SIM when class information is required. The class information is stored in the binding information of the code file.

If the subprogram declares the query variable in the global declarations, the database declaration must precede the query variable declaration.

If the query variable is associated with a DMRECORD, the DMRECORD must be declared before the query variable. Binder verifies that the host program and the subprogram query variables reference the same database class or DMRECORD.

Referencing a SIM Database

You must declare a SIM database in the host program and in the subprogram. When the compiler encounters the database declarations, it generates a SIM library template in the outer block of the host program and generates a SIM library template in the subprogram. These templates import all the library objects in the SIM system.

Binder changes all code references of the SIM library objects in the subprogram to match the SIM library objects in the host program. The following example shows the SIM library template generated by the compiler and the SIM library objects in the subprogram that Binder will change to match those of the host program.

Example

Host Program H1

```
      BEGIN
      SEMANTIC DATABASE UNIVDB:(INSTRUCTOR,STUDENT);

   (2,2) = FUNNY SIRW                  %  These lines of code for
   (2,3) = SUPPORT LIBRARY TEMPLATE    %  the SIM library template
   (2,4) = LIBRARY TEMPLATE MARKER     %  are generated by the
                                       %  compiler.
   (2,5) = SUPPORT LIBRARY PROCEDURE   %
         .                             %  Several support library
         .                             %  procedures generated by
         .                             %  the compiler are bound.
  (2,15) = SUPPORT LIBRARY PROCEDURE   %
         .                             %  Other data structures
         .                             %  are generated by the
         .                             %  compiler and placed here.

      PROCEDURE REPLACE_ME (R);
   (2,1B) = REPLACE_ME
         REAL R;
         EXTERNAL;

      OPEN UNIVDB;

            % Additional program statements
            % could be included here.
      DELETE STUDENT WHERE CURRENT(STUQ) = STU;
      REPLACE_ME (10);
      CLOSE UNIVDB;
      END.
```

Subprogram S1

```
      [REAL I, J;                                      % Global
       SEMANTIC DATABASE UNIVDB:(INSTRUCTOR,STUDENT);] % declaration

   (2,4) = FUNNY SIRW                  %  These lines of code for
   (2,5) = SUPPORT LIBRARY TEMPLATE    %  the SIM library template
```

```
(2,6) = LIBRARY TEMPLATE MARKER       %  are generated by the
                                      %  compiler.
(2,7) = SUPPORT LIBRARY PROCEDURE     %
      .                               %  Several support library
      .                               %  procedures generated by
      .                               %  the compiler are bound.
(2,17) = SUPPORT LIBRARY PROCEDURE    %

    PROCEDURE REPLACE_ME (R1);
(2,1D) = REPLACE_ME
       REAL R1;
       BEGIN
       DELETE INSTRUCTOR WHERE SALARY > R1;
       END;
```

In the preceding example, the host program, H1, declares a SIM database in the outer block and an external procedure, REPLACE_ME, to be bound. The compiler builds the SIM library template.

The subprogram S1 declares REPLACE_ME, which references the database. The compiler builds the SIM library template for the subprogram. However, the stack locations in the subprogram for the SIM library objects and the procedure do not match the stack locations for those elements in the host program. Binder fixes these code references in the subprogram so that they match those in the host program. For example, Binder changes the stack location (2,D) of the procedure REPLACE_ME in the subprogram to match the stack location (2,B) of the procedure REPLACE_ME in the host program.

Referencing a SIM Entity Reference Variable in a Host Program

In this example, the subprogram references an entity reference variable declared in the host program. The database must be declared before the entity reference declaration in the global declarations of the host program.

Example

Host Program H2

```
     BEGIN
     SEMANTIC DATABASE UNIVDB:(STUDENT);
     ENTITY REFERENCE STU_REF (STUDENT);
  (2,B) = STU_REF

     PROCEDURE REPLACE_ME (R);
  (2,E) = REPLACE_ME
        REAL R;
        EXTERNAL;

     <rest of declarations>

     OPEN UNIVDB;

     <program statements>

     DELETE STUDENT WHERE CURRENT(STUQ) = STU_REF;
     REPLACE_ME (10);
     CLOSE UNIVDB;
     END.
```

Subprogram S2

```
     [REAL I, J, K;                              % Global declaration
      SEMANTIC DATABASE UNIVDB:(STUDENT);        %
      ENTITY REFERENCE STU_REF (STUDENT);]       %
  (2,D) = STU_REF

      PROCEDURE REPLACE_ME (R1);
  (2,10) = REPLACE_ME
         REAL R1;
         BEGIN
         DELETE STUDENT WHERE CURRENT(STUQ) = STU_REF;
         END;
```

In this example, the host program, H2, declares the entity reference variable STU_REF. The compiler determines whether STUDENT is a valid database class and, if so, allocates the proper number of stack cells for it. The compiler also supplies class and size information about STU_REF in the binding information.

Binder also verifies that STU_REF references the same class in the same database in both the host program and the subprogram. As an added check, Binder verifies the size of

STU_REF. Binder fixes the code references in the subprogram so that all code references to STU_REF match those of the host program.

Referencing a SIM Query Variable in a Host Program

In this example, the subprogram references a query variable declared in the host program. The database and an optional DMRECORD must be declared before the query variable declaration in the global declarations.

Example

Host Program H3

```
     BEGIN
     SEMANTIC DATABASE UNIVDB:(STUDENT);
     QUERY STUQ (STUDENT);
  (2,B) = STUQ

     PROCEDURE REPLACE_ME (R);
  (2,C) = REPLACE_ME
        REAL R;
        EXTERNAL;
     <rest of declarations>

     OPEN UNIVDB;

     <program statements>

     DELETE STUDENT WHERE CURRENT(STUQ) = STU_REF;
     REPLACE_ME (10);
     CLOSE UNIVDB;
     END.
```

Subprogram S3

```
     [REAL I, J, K;                          % Global declaration
      SEMANTIC DATABASE UNIVDB:(STUDENT);    %
      QUERY STUQ (STUDENT);]                 %
  (2,D) = STUQ

      PROCEDURE REPLACE_ME (R1);
  (2,10) = REPLACE_ME
         REAL R1;
         BEGIN
         DELETE STUDENT WHERE CURRENT(STUQ) = STU_REF;
         END;
```

In the preceding example, the host program, H3, declares the query variable STUQ. The compiler determines whether STUDENT is a valid database class and, if so, supplies information about STUQ in the binding information. The compiler verifies that STUQ declared in the host program and in the subprogram references the same class (or DMRECORD) in the same database.

Binder fixes the code references in the subprogram so that all code references to the global query variable STUQ match those of the host program.

Adding Query Variables as New Globals

The following example illlustrates how a query variable that does not exist in the host program can be declared in the global declarations portion of the subprogram. A database declaration must precede the query variable declaration. If the query variable is associated with a DMRECORD, the DMRECORD must be declared before the query variable.

When a query variable is declared in this way, Binder adds the query variable as a new global item and alters all subprogram code references to the query variable to match the host program code references to that query variable. Locally declared query variables are unaffected by Binder.

Example

Host Program H8

```
     BEGIN
     SEMANTIC DATABASE UNIVDB;

     PROCEDURE REPLACE_ME (R);
  (2,B) = REPLACE_ME
        REAL R;
        EXTERNAL;

     <remaining declarations>

     OPEN UNIVDB;

     <program statements>

     REPLACE_ME (10);
     CLOSE UNIVDB;
     END.
```

Subprogram S8

```
     [REAL I, J, K;                      %Global declaration
      SEMANTIC DATABASE UNIVDB;
      QUERY STUQ (STUDENT);]
  (2,D) = STUQ
      PROCEDURE REPLACE_ME (R1);
  (2,E) = REPLACE_ME
         REAL R1;
         BEGIN
         DELETE STUDENT WHERE CURRENT (STUQ) = STU_REF;
         END.
```

In the preceding example, the subprogram, S8, declares the query variable STUQ. STUQ is a SIM construct used for querying database information, which is the database class

STUDENT in this example. The compiler determines if STUDENT is a valid database class and supplies information about STUQ in the binding information. The compiler also allocates STUQ as a new global for the host. Binder alters the subprogram code references to the global query variable STUQ to match the host program code references to that query variable.

Referencing a SIM Database in a Pascal Host

To create a host program, the Pascal program must declare an external module. The variables, procedures, functions, and databases of the host program become visible to an external subprogram if the following occurs:

- These items are exported by host modules before the declaration of the external module.
- These items are imported by the EXTERNAL module.

Refer to the *Pascal Reference Manual*, *Volume 1* for more information about compiling modules in the Pascal host program.

The following example shows a Pascal host program that accesses a SIM database. Appropriate Pascal syntax is used to enable modules to bind external modules (subprograms) written in other languages. The Pascal compiler creates a file titled, BINDERINPUT, which contains Binder instructions to bind the external modules.

Two modules are declared in the example program: data_access and data_user. The data_access module defines the database and the SIM variables used in the program. The SIM variables are exported, which makes them available to other modules such as data_user.

The data_user module is declared external. The export list of this module contains an ALGOL subprogram, named ALGOL_subroutine, to be bound. The data_user module imports all the interface identifiers from the data_access module, including the database and other variables. This makes the database and the variables visible to the external program.

The implementation section of the data_access module imports the ALGOL subprogram, ALGOL_SUBROUTINE, and uses it in a function call.

Example

Host Program H4

```
MODULE DATA_ACCESS INTERFACE
    (univdb: database,
     Mydict: DICTIONARY <FUNCTIONNAME = '39DATADICTIONARY'>);
export data_access (intq, stuq, univdb, ent_ref, dmrec, dostuff);
from univdb import instructor, student;
type stu_rec_type = record
     stu_no : integer;
     end;
  stu_rec = dmrecord (stu_rec_type);
var  intq    : query (instructor);
     stuq    : query (student);
     ent_ref : entityreference (student);
     dmrec   : stu_rec;
```

```
procedure dostuff;
end;

module data_user external;
export ALGOL_external (ALGOL_subroutine);
import data_access;
function ALGOL_subroutine: integer;
end;

module data_access implementation;
import ALGOL_external;
var salary_increase  : Boolean;
     1imres           : dmstatetype;
     int              : integer;

procedure dostuff;
var i : integer;
begin
   i := ALGOL_subroutine;
   open(univdb, update);
   begintransaction;
        startinsert (intq);
   If salary_increase then
        assign (intq.salary, 50000);
   applyinsert (intq);
   close (univdb);
end;                                    {End DOSTUFF}
end.                                    {Module implementation}

program p;                              {Main program}
import data_access;
   begin
     dostuff;
   end;
```

Section 8
Printing Binding Information

You can compile a program so that it generates the binding information used to bind the code file to another code file. Binding information consists of a description of the elements in the code file, such as

- The lex level and code segment location for each procedure
- A description of the items in the local directory of each procedure, including variables and arrays and their characteristics
- A description of the information in the global directory of a procedure
- A description of the information in an external procedure
- The identification of various other elements, including the block exit pointer, the first executable code segment, and the global stack size

Generating Binding Information

Language compilers differ slightly in the instructions they require to generate binding information. These differences are described in the following list:

- ALGOL and FORTRAN

  A program compiled in ALGOL or FORTRAN will have binding information generated when the NOBINDINFO compiler control option is set to FALSE. The default setting is FALSE.

- COBOL

  A program compiled in COBOL will have binding information generated if any of the following conditions exist:

  – Its lexical (lex) level is greater than 2.

  – It contains a procedure declared as EXTERNAL.

  – The BINDINFO compiler control option is set to TRUE.

- FORTRAN77

  A program compiled in FORTRAN77 will have binding information generated when the BINDINFO compiler control option is set to TRUE.

- Pascal

  A program compiled in Pascal will have binding information generated when a module is declared as external in the program.

Using the PRINTBINDINFO Utility

You can print an analysis of the binding information of a bound or unbound code file by using a utility named SYSTEM/PRINTBINDINFO (hereafter referred to as the PRINTBINDINFO utility). The binding information for each separate procedure of a multiprocedure library file (an ALGOL, FORTRAN, or FORTRAN77 program compiled with the LIBRARY option set to TRUE) is analyzed and printed. A list of the identifiers in the separate procedures is written at the beginning of the printed output.

You can start the PRINTBINDINFO utility from a WFL job or from a CANDE session.

The WFL syntax for running PRINTBINDINFO is as follows:

```
? BEGIN JOB PRINT/BINDER/INFO;
  RUN SYSTEM/PRINTBINDINFO;
  FILE CODE = <code file title>;
  FILE LINE = <line printer output file title>;
? END JOB.
```

The CANDE syntax for running PRINTBINDINFO is as follows:

```
RUN $SYSTEM/PRINTBINDINFO; FILE CODE = <code file title>;
FILE LINE = <line printer output file title>
```

The <code file title> and <line printer output file title> constructs are used in both the WFL and the CANDE syntaxes.

The <code file title> construct specifies the code file whose binding information is to be analyzed. Its default file characteristics are as follows:

```
KIND=PACK, FAMILYNAME="DISK.", FILETYPE=8, INTMODE=SINGLE
```

The <line printer output file title> construct specifies the output file created by PRINTBINDINFO when the LIST option is set. Its default file characteristics are as follows:

```
KIND=PRINTER, INTMODE=EBCDIC, MAXRECSIZE=22
```

Note: *If you try to run PRINTBINDINFO on a code file that does not contain binding information, the system generates an error message and terminates execution.*

Example

Consider the following ALGOL program:

```
BEGIN
    INTEGER I;
    REAL ARRAY ARY[0:4,0:9];
    REAL PROCEDURE RP(A);
        VALUE A; BOOLEAN A;
    BEGIN
        INTEGER J;
```

```
    END RP;
END.
```

If this program is compiled and its code file is given the title OBJECT/EXAMPLE/1, then the following CANDE command can be used to run PRINTBINDINFO to print a complete analysis of the binding information of OBJECT/EXAMPLE/1:

```
RUN $SYSTEM/PRINTBINDINFO; FILE CODE = OBJECT/EXAMPLE/1
```

The output produced by PRINTBINDINFO appears as follows:

```
PROGRAM DESCRIPTION:


     PROCEDURE DIRECTORY    ****************************************
           PROCEDURE BLOCK#1;   LEX LEVEL: H02; CBIT  CODE SEGMENT H0003
               LOCAL DIRECTORY
0001               VARIABLE (INTEGER)             H(02,0002)   I
0002               ARRAY (REAL)                   H(02,0003)   ARY
                      NUMBER OF DIMENSIONS: 02
0003               FUNCTION (REAL)                H(02,0004)   RP
                      PARAMETERS
                          NUMBER OF PARAMETERS: 01
                          VARIABLE (BOOLEAN)
                      LIT48 POINTER FOR MAKING PCW: H(0003:0007:3,LL=00)
               PROCEDURE RP;   LEX LEVEL: H03;        CODE SEGMENT H0005
                   LOCAL DIRECTORY
0004                  VARIABLE (REAL)             H(03,0003)   RP.VALUE
0005                  VARIABLE (BOOLEAN)          H(03,0002)   A
0006                  VARIABLE (INTEGER)          H(03,0004)   J
     END OF PROCEDURE DIRECTORY    ************************************

     GLOBAL DIRECTORY  ****************************
0007     INTRINSIC (REAL)                 H(01,0004)  ?007
0008     INTRINSIC (REAL)                 H(01,0006)  ?010
     END OF GLOBAL DIRECTORY ************************

     BLOCK EXIT POINTER: H(0003:0006:2, LL=02)
     FIRST EXECUTABLE CODE: H(0003:0000:1, LL=02)
     POINTER TO END OF D2 STACK: H(0003:0009:0, LL=00)
     GLOBAL STACK SIZE: 5
     SOFTWARE CONTROL WORD IMAGE: H800000001000



NO EXTERNAL PROCEDURES.
```

Printing Binding Information for Specific Procedures

You can select certain procedures and blocks, and items within those procedures and blocks for which you want to print the binding information. You make selections by using a SELECTIDS file.

The SELECTIDS file consists of a list of one or more EBCDIC identifiers separated by one or more blanks. If a SELECTIDS file is present when PRINTBINDINFO is run, binding information is analyzed and printed for only the listed items.

If an identifier appears in the SELECTIDS file, information about that identifier is printed only if one of the following conditions is true:

- The identifier belongs to a procedure or block in the program.
- The identifier is described in the program description outside the global directory and the own directory.
- The identifier is described in the global directory.
- The identifier is described in the own directory.
- The identifier is described in the local directory of a procedure or block, and the identifier of that procedure or block also appears in the SELECTIDS file.

For example, if identifier M is declared in procedure P, then information about M is printed only if both M and P appear in the SELECTIDS file.

If identifier J is declared in the outer block of an ALGOL program, then information about J is printed only if both J and the identifier of the outer block appear in the SELECTIDS file.

(The identifier of the outer block of an ALGOL program is *BLOCK#1* for programs compiled with Mark 3.5 and later compilers and *B.0000* for programs compiled with compilers earlier than Mark 3.5.)

The default characteristics for the SELECTIDS file are as follows:

```
KIND=READER, INTMODE=EBCDIC, FILETYPE = 8
```

To use a disk file for SELECTIDS, specify KIND=DISK when file-equating.

Example

The following WFL job runs PRINTBINDINFO to analyze OBJECT/EXAMPLE/1, the ALGOL program shown in the previous example, but restricts the analysis by providing a SELECTIDS file:

```
? BEGIN JOB RUN/PRINTBINDINFO;
      RUN SYSTEM/PRINTBINDINFO;
      FILE CODE = OBJECT/EXAMPLE/1;
      DATA SELECTIDS
      BLOCK#1
      ARY
```

```
      J
 ? END JOB.
```

The output produced by this job appears as follows:

```
SELECTED IDENTIFIERS:

BLOCK#1
ARY
J



PROGRAM DESCRIPTION:

     PROCEDURE DIRECTORY    ****************************************
           PROCEDURE BLOCK#1;   LEX LEVEL: H02;  CBIT CODE SEGMENT H0003
               LOCAL DIRECTORY
0001               ARRAY (REAL)                    H(02,0003)   ARY
                      NUMBER OF DIMENSIONS: 02
     END OF PROCEDURE DIRECTORY    ************************************

     GLOBAL DIRECTORY  ******
     END OF GLOBAL DIRECTORY ******



NO EXTERNAL PROCEDURES.
```

In this output, no information is printed for J because J is described in the local directory of the procedure RP, and RP does not appear in the SELECTIDS file. Information about ARY was printed because ARY appears in procedure BLOCK#1, and BLOCK#1 appears in the SELECTIDS file.

Output Options

You can use the following three options to affect the format of the output from the PRINTBINDINFO utility. To enable one or more of these options, you must assign a negative value to the TASKVALUE attribute. To do this, set bit 46 of the TASKVALUE attribute to 1. In addition, you must set a bit for each specific option as indicated below. A list of the enabled options appears at the beginning of the printed output.

DEBUG	Prints binding information in unanalyzed as well as analyzed form. To enable the DEBUG option, set bit 0 (zero) of the TASKVALUE attribute to 1.
IGNORELOCALDIR	Prevents local directories from being analyzed and printed. To enable the IGNORELOCALDIR option, set bit 1 of the TASKVALUE attribute to 1.
NOREFERENCES	Prevents code references from being analyzed and printed. To enable the NOREFERENCES option, set bit 2 of the TASKVALUE attribute to 1.

Example

The following CANDE command causes PRINTBINDINFO to analyze the code file, OBJECT/TEST, with the options IGNORELOCALDIR and NOREFERENCES enabled:

```
RUN $SYSTEM/PRINTBINDINFO; VALUE=-6; FILE CODE = OBJECT/TEST
```

Appendix A
Warning and Error Messages

This appendix contains an alphabetical listing of the warning and error messages that you might encounter when using Binder and provides corrective action when applicable.

A COMPILER ERROR WAS DETECTED AT BINDER LINE NUMBER nnnnnnnn

- Refer this problem to your Customer Service Representative.

A <DIRECTORY SPECIFIER> IS NOT ACCEPTABLE HERE

- This error results when a directory specifier appears in a HOST statement.

A <FILE NODE> WAS EXPECTED IN THIS FILE NAME

- There is an error in the format of the file name.

A GLOBAL VARIABLE (THAT WAS REFERENCED FROM AN INTRINSIC BEING BOUND) COULD NOT BE FOUND. USE A BIND STATEMENT

- An intrinsic being bound to an intrinsic file references a global variable that
  - Is not an MCP global item
  - Is not initialized to a correct address couple by an INITIALIZE statement
  - Does not already exist in the intrinsic file
    - Is not specified to be bound on a BIND statement

A LEFT PARENTHESIS IS MISSING HERE

- In an INITIALIZE statement, the left parenthesis at the beginning of the address couple is missing.

A NEW GLOBAL VARIABLE MAY NOT BE ADDED TO A HOST THAT IS A SUBPROGRAM WITH NO GLOBAL DECLARATIONS

- The host program is a subprogram that contains no global declarations. Binder does not allow a new global to be added to such a host in the course of binding a nested subprogram.

A QUOTE MARK WAS EXPECTED

- An identifier that begins with a quotation mark is missing the ending quotation mark.

A RIGHT PARENTHESIS WAS EXPECTED HERE

- This error is given in the following situations:
  - In an INITIALIZE statement, the right parenthesis at the end of the address couple is missing.
  - In a BIND statement of the form BIND <intrinsic specification>, the right parenthesis at the end of the language list is missing.

  - In a file specifier or directory specifier, the right parenthesis following the usercode is missing.

A SEMICOLON WAS EXPECTED HERE

- The semicolon (;) at the end of the Binder input statement is missing.

A STATEMENT "USE <HOST-FILE-LIBOBJECT-NAME>" FOR <SUBPROGRAM-LIBOBJECT-NAME> IS REQUIRED

- A USE statement is required for one of the following reasons:
  - A library in the host program specified an alias for the name of a library object and a subprogram made reference to the actual name of the same library object.
  - The host program used the actual name for the library object and the subprogram used an alias.
  - The host program and the subprogram used different aliases for the same library object.
- The library object names that you include in the USE statement must reflect the name by which the library object is referenced in the respective programs, regardless of whether an alias has been assigned. For example, assume that the host file declares a library object within a library named "ALIASED" and specifies that "ALIASED" refers to "LIBRARYOBJECT1." If a subprogram refers directly to "LIBRARYOBJECT1," you must provide the following USE statement: *USE ALIASED FOR LIBRARYOBJECT1*.

A SUBPROGRAM IDENTIFIER WAS EXPECTED HERE

- In a BIND statement, the word BIND is not followed by an identifier or an equal sign (=).

A VALID INTEGER WAS EXPECTED HERE

- This error is given in the following situations:
  - In an INITIALIZE statement, either the first or second number of the address couple is not a valid integer.
  - In a BIND statement of the form BIND <intrinsic specification>, either the first or second number of the intrinsic number pair is not a valid integer.

A VALID LANGUAGE IDENTIFIER WAS EXPECTED HERE

- In a BIND statement of the form BIND <intrinsic specification>, an item in the language list is not a valid language identifier.

AN ARRAY PARAMETER MUST BE DECLARED BEFORE THE 24TH PARAMETER

- The procedure to be bound has more than 24 parameters, and an array was discovered after the 24th parameter.
- Avoid this error by declaring arrays within the first 24 parameters.

AN ARRAY THAT WAS ADDED AS A NEW GLOBAL VARIABLE HAD NO LENGTH SPECIFIED FOR IT

- The array that was added as a new global array to the host program had no length specified for it.

In ALGOL, this results from not declaring an upper bound for the array within the brackets used for declaring such global items for separate compilation.

In COBOL, this message occurs when a new array is added to a host program by Binder. New global arrays are not allowed for COBOL binding.

AN ENTRY POINT CANNOT BE ADDED AT OTHER THAN THE GLOBAL LEVEL

- If a FORTRAN subprogram containing entry points is compiled at a lexical level higher than 3 and bound to a host program in which one of the entry point variables was not previously declared, an error results. The entry point would have to be added at the global level, which is incompatible with its execution level.
- When binding a higher level subprogram containing entry points, declare all entry points directly within the program unit to which the subprogram is bound.

AN INTERNAL BINDER ERROR HAS OCCURRED

- Refer this problem to your Customer Service Representative.

AN INTERNAL BINDER ERROR HAS OCCURRED—THE PROCEDURE DIRECTORY AND THE INFO TABLE ARE MISMATCHED

- Refer this problem to your Customer Service Representative.

BINDER CONTROL OPTIONS MAY NOT APPEAR IN THE MIDDLE OF A BINDER STATEMENT

- Binder control records can appear between Binder statements but cannot appear within a Binder statement contained on more than one input record.

BINDER_MATCH OPTION MISMATCH: THE HOST WAS COMPILED FOR POSIX AND THE SUBPROGRAM <title> WAS NOT COMPILED FOR POSIX

- The host object file is C code that contained either *#define _POSIX_SOURCE* or *#define _ASERIES_SOURCE 410* /* or larger value*/. The subprogram <title> is C code that did not contain either define or contained *#define _ASERIES_SOURCE 409* /* or smaller value*/. C code being bound into a C host that was compiled for POSIX must have either of the first two defines above.

BINDER_MATCH OPTION MISMATCH: THE HOST WAS NOT COMPILED FOR POSIX AND THE SUBPROGRAM <title> WAS COMPILED FOR POSIX

- The subprogram <title> is C code that contained either *#define _POSIX_SOURCE* or *#define _ASERIES_SOURCE 410* /* or larger value*/. The host file is C code that did not contain either define or contained *#define _ASERIES_SOURCE 409* /* or smaller value*/. C code compiled for POSIX can only be bound into a C host that must have either of the first two defines above.

BINDER_MATCH OPTION '<name>' IS DIFFERENT: FILE <title1> HAS VALUE '<value1>' BUT FILE <title2> HAS VALUE '<value2>'

- BINDER_MATCH options are generated when you set the $BINDER_MATCH compiler option in ALGOL, C, and COBOL85. This option also can be generated automatically by the C and COBOL85 compilers. This message indicates that Binder detected two BINDER_MATCH options that have the same name string but different value strings. The two strings must match exactly, including casing and blank characters.

- Review the source code of the files you are attempting to bind and ensure that the same compile-time BINDER_MATCH options have the same values. Then execute Binder again.

BOUND CODE LEVEL CHANGED FROM rr.lll TO rr.lll.

- This error message is given in the following situations:
  - You tried to bind a subprogram compiled with an earlier version of the compiler than that used to compile the host program or previously bound subprograms.
  - Binder is an earlier version than the bound programs.

The bound code file is set to the earliest version found.

COMMA EXPECTED

- This error message is given in the following situations:
  - In an INITIALIZE statement, the comma in the address couple is missing.
  - In a BIND statement of the form BIND <intrinsic specification>, the comma after the first integer of the intrinsic number pair is missing.

DUE TO THE ABOVE ERROR(S), THE BINDING OF THIS PROCEDURE IS DISCONTINUED

- The definition of a subprogram within a subprogram file was found to be incompatible with the subprograms definition in the host. The reason for the incompatibility was indicated by the error messages emitted prior to this message.

  Binder discontinues binding the procedure at this point, resets the error count back to the value it had before the start of binding the subprogram, and continues the binding process. The given subprogram is treated as if no attempt had been made to bind it.

  If two subprograms within the host program are known by the same identifier and Binder attempts to bind both occurrences of the identifier from the same subprogram, the definition of the separate subprogram probably would be incompatible with one of the occurrences, but might be compatible with the other occurrence. Thus, the subprogram would be bound correctly to its compatible occurrence, and the incompatible occurrence would not affect the bind in an adverse way. This result might have been the original intention of the user who did not realize that a subprogram identifier occurred twice within the host.

EITHER THE SUBPROGRAM MUST BE COMPILED WITH A SSR 42.3 OR LATER COMPILER OR THE HEAP ROW SIZE OF ALL SUBPROGRAMS MUST MATCH THAT OF THE HOST.

- The size of a row in the heap is determined by the value of the MEMORY MODEL option and the value of the LONGLIMIT option when the memory model is TINY or LARGE. The heap row size in one or more of the subprograms that preceded the current subprogram in the bind is different from that of the host. The current subprogram was compiled with a pre-SSR 42.3 compiler. Either the current subprogram must be recompiled with a SSR 42.3 or later compiler or the subprograms with a memory model or possibly a long limit different from the host must be recompiled with the host memory model and long limit.

EITHER THE SUBPROGRAM MUST BE COMPILED WITH A SSR 42.3 OR LATER COMPILER OR THE MEMORY MODEL OF ALL SUBPROGRAMS MUST MATCH THAT OF THE HOST.

- The value of the MEMORY MODEL option in one or more of the subprograms that preceded the current subprogram in the bind is different from that of the host. The current subprogram was compiled with a pre-SSR 42.3 compiler. Either the current subprogram must be recompiled with a SSR 42.3 or later compiler or the subprograms with a memory model different from the host must be recompiled with the host memory model.

FAR/NEAR SPECIFICATION DOES NOT MATCH BETWEEN SUBPROGRAMS

- A C pointer variable is declared as far in one subprogram, but as near in another. The module declaring the pointer as near can misinterpret the pointer if the other module assigns a far pointer value to the variable.

FILE <FILENAME> NOT AVAILABLE

- A *BIND ? FROM <file>* statement was issued and Binder did not find the file.

FORTRAN77 SUBPROGRAMS MAY NOT BE BOUND INTO A MARK 3.6 FORTRAN77 HOST

- Compile the host program with a Mark 3.7 level or newer compiler.

<identifier 1> DECLARED IN SUBPROGRAM AS <CONNECTION or STRUCTURE> BLOCK TYPE <identifier 2> BUT DECLARED IN HOST AS <CONNECTION or STRUCTURE> BLOCK TYPE <identifier 3>

- The name of the structure or connection block type for <identifier 1> in the subprogram is different from its name in the host program. <Identifier 1> is the name of a:

– Structure block array

– Structure block reference

– Structure block variable

– Single connection library

– Multi-connection library

– Connection block reference

- <Identifier 2> is the name of the structure or connection block type in the subprogram.
- <Identifier 3> is the name of the structure or connection block type in the host program.
- The type name in the subprogram must be the same as the type name in the host program.

IN A CODE FILE THAT CANNOT RUN ON ANY MACHINE, A COMMON BLOCK CANNOT BE EXTENDED BECAUSE IT IS CODEFILE INITIALIZED

- The FORTRAN77 host program was compiled with CODEFILEINIT set, and locations in the common block were initialized. A subprogram being bound declared the common block to be longer than the common block in the host. Because the initial value of the common

block was initialized within the code file, the common block could not be extended.

IN THIS BIND STATEMENT, AN EQUAL SIGN WAS EXPECTED HERE

- In this statement, the equal sign after the identifier is missing.

<lo> <LIBRARY OBJECT> REQUIRES LIBRARY <ll>

- Binder did not find the library <ll> in the host program, so it did not bind the library object <lo>. This error can occur if the library names are different in the host program and the subprogram.
- To match different names, include a USE statement in the primary input file. For the syntax and explanation of the USE statement, see Section 3.

  Refer this problem to your Customer Service Representative if the preceding solution is not applicable.

MARK nn CODE FILES MAY NOT BE BOUND; ONLY MARK mm, OR LATER CODE FILES MAY BE BOUND

- You tried to bind a code file that was more than three system software releases old. The letters nn indicate the release level of the code file. The letters mm indicate the earliest release level of software that can be used with Binder, which is software that is three releases older than the current level of Binder.

MEMORY MODEL MISMATCH: THE HOST USES THE nnnnnnnn MODEL, THE SUBPROGRAM USES THE nnnnnnnnn MODEL.

- The value of the MEMORY_MODEL option must be the same for a C host program and a C subprogram when (1) binding object code files compiled with a C compiler before SSR 42.3, or (2) the compiler control option FARHEAP is false.

<NAME> EXPECTED

- This error is given when the following situations occur within a file specifier or directory specifier:
  - The usercode is not a valid name.
  - The family name is not a valid name.
  - The specifier does not begin with a valid name or the equal sign.
  - A right parenthesis, an asterisk, or a slash is not followed by a valid name or an equal sign.
  - A name is specified to be two quotation marks with no characters in between.

NEW GLOBAL AND <OWN> VARIABLES CANNOT BE ADDED TO THE HOST

- If the host is a NEWP program, new global variables and own variables cannot be added.

NEW GLOBAL VARIABLES CANNOT BE ADDED WHILE BINDING SEGMENT 1 OF THE MCP

- While MCP segment 1 is being bound, new globals cannot be added to the MCP.

ONLY MULTIPROCEDURE FILES ARE ALLOWED IN UNIVERSAL BIND STATEMENT

- One of the following two types of statements was given as input to Binder:

```
BIND = FROM A/B
BIND P, Q, SUBR FROM A/B;
```

  In these examples, A/B is not a library or multiprocedure file. Because A/B contains only one subprogram, it should not be used in the above BIND statements.

ONLY THE LAST BIND STATEMENT ENCOUNTERED WILL BE USED

- If more than one BIND statement is found, the last statement entered is used.

OUTPUTMESSAGE ARRAY NAMES MUST BE UNIQUE THROUGHOUT THE ENTIRE PROGRAM

- Output message array names are an exception to the rules of scope for an identifier. They must be unique throughout the entire program.

PL/I PROGRAMS MAY ONLY BE BOUND TO PL/I PROGRAMS

- A subprogram compiled by the PL/I compiler can be bound only to host programs compiled by the PL/I compiler.

REPLACEMENT BINDING IS NOT ALLOWED

- If the host program is a NEWP program, only output message arrays or procedures declared as EXTERNAL can be bound.

SINCE NO INTRINSIC NUMBER WAS GIVEN, THIS INTRINSIC CANNOT BE REFERENCED OUTSIDE OF THE INTRINSICS FILE

- You did not specify an intrinsic number pair for a new intrinsic being added to an intrinsic file. The intrinsic can be called by other intrinsics within the file, but cannot be invoked from a user program.

THE AREASIZE OF THE BOUND CODE FILE IS TOO SMALL. INCREASE IT BY SETTING THE AREASIZE FILE ATTRIBUTE DURING THE BIND

- During the conclusion of intrinsic binding, Binder found that the area size of the bound code file was smaller than required.
- Increase the area size of the bound code file by using the AREASIZE file attribute as shown in the following example:

```
BEGIN JOB MAKE/INTRINSICS;
  BIND NEW/INTRINSICS WITH BINDER LIBRARY;
    BINDER FILE CODE (AREASIZE = 2016);
BINDER DATA
$ SET INTRINSICS
        BIND = FROM INTR/=;
        BIND MYINT = 101,1 (ALGOL, FORTRAN) FROM INTL/=;
? ! END DATA
END JOB.
```

THE BIND STATEMENT FOR THIS PROCEDURE WAS NOT USED --(EITHER THE ABOVE BIND STATEMENT(S) WERE OVERRIDDEN BY ANOTHER

STATEMENT, OR THE PROCEDURE IDENTIFIER DID NOT EXIST IN THE HOST AND WAS NOT CALLED BY ANY PROCEDURE BOUND IN.)

- If a subprogram identifier specified in a BIND statement is not declared in the host program or otherwise encountered during the binding process, the subprogram is not bound.

THE BINDER OPTION DEBUG HAS BEEN DEIMPLEMENTED. INSTEAD USE THE TEST AND DEBUG SYSTEM (TADS)

- The Binder DEBUG option is no longer valid.
- Use the Test and Debug System (TADS) appropriate to the program in error to identify the binding problem.

THE BINDER WAS UNABLE TO BIND ONE OR MORE PROCEDURES BUT THE CODE FILE IS STILL EXECUTABLE

- This warning message is given when Binder has been unable to bind one or more of the procedures declared as EXTERNAL or explicitly named in a BIND statement. The bound code file can be executed. However, if an attempt is made to execute an external subprogram that was not bound, the following error message is given:

  ```
  <identifier> NOT BOUND
  ```

  If Binder is unable to replacement bind a subprogram, the original subprogram remains in the bound code file.

THE BINDER'S INTERNAL CONSTANT ARRAY HAS OVERFLOWED—THE BINDER'S CAPACITY HAS BEEN EXCEEDED

- Refer this problem to your Customer Service Representative.

THE BINDER'S INTERNAL INFO TABLE HAS OVERFLOWED—THE BINDER'S CAPACITY HAS BEEN EXCEEDED

- Refer this problem to your Customer Service Representative.

THE CODEFILES CONTAIN DATA MANAGEMENT LEVELS THAT ARE INCOMPATIBLE AND CANNOT BE BOUND

- This error message refers to the binding of DMSII databases. The host program and all subprograms that reference a DMSII database must all be compiled with the same level of DMSII software.

THE COMBINED SPACE IN THE HEAP REQUIRED FOR DATA ITEMS DECLARED IN THE OUTER BLOCK (nnnn WORDS) EXCEEDS THE MAXIMUM AVAILABLE SPACE OF nnnn WORDS.

- The global variables require more space than can be supported with the current value of the MEMORY MODEL option. Either reduce the size requirements of the LONGLIMIT option or use a larger memory model.

THE COMMON BLOCK CANNOT BE EXTENDED FOR THIS HOST. YOU MUST RECOMPILE THE HOST

- A subprogram tried to extend a global array by declaring it larger in the subprogram than in the host. The new size was too large to fit in the array declaration parameters of the host. This situation can occur only with hosts compiled before release 3.6.

THE DECLARATION IN SUBPROGRAM MUST BE COMPATIBLE WITH THE DECLARATION IN HOST

- The description of a library object in the subprogram is not compatible with the description of the same library object in the host program. This incompatibility can occur with mismatched parameter types or with mismatched by-reference or by-value usage.

THE FORTRAN77 HEAP VECTOR EXCEEDS MAXIMUM LENGTH. RECOMPILE THE SEPARATE FILE WITHOUT THE "HEAP" OPTION

- New heap vector entries in a FORTRAN77 subprogram would make the length of the heap vector exceed its maximum of 65,535.

THE HOST AND THE SUBPROGRAM DO NOT HAVE THE SAME LIBRARY SHARINGCLASS

- This error occurs if the subprogram and the host program being bound are libraries, but have a mismatched SHARINGCLASS. For example, the subprogram is a share-by-all library, whereas the host program is a private library.

THE HOST CODE FILE WAS PRODUCED BY A PREVIOUS BIND. ADDITIONAL BINDING IS NOT ALLOWED

- A bound C program cannot be used as the host of a subsequent bind.

THE HOST FILE IS NOT AN INTRINSICS FILE

- The INTRINSICS option has the value TRUE in Binder, but the host program is not an intrinsics file.

THE HOST WAS COMPILED AT A LEXICAL LEVEL TOO HIGH

- Compile the host program at lexical level 2 or 3.

THE HOST WAS COMPILED WITH THE CONCURRENT EXECUTION OPTION SET, BUT THE SUBPROGRAM WAS NOT

- Binder issues this warning when trying to bind a subprogram that doesnt have the CONCURRENTEXECUTION option set into a host program that was compiled with the option set. In general, concurrent execution of multiple tasks requires all subprograms to be compiled with the option set. The only exception is a subprogram that runs only while there is no other execution stream, such as before a shared-by-all library freezes.
- Recompile the subprogram with the CONCURRENTEXECUTION option set, and then rerun Binder.

THE IDENTIFIER OF THE SEPARATE PROCEDURE DOES NOT MATCH THE DECLARATION IN THE HOST

- Binder was directed by a BIND statement to bind the given subprogram from a specific file. The subprogram identifier in the subprogram file does not match the declaration in the host. Binder generates this message and creates a USE statement that matches the two identifiers. Note that this situation cannot occur when binding from a specific library file or multiprocedure file.

THE INITIALIZE STATEMENT IS LEGAL FOR INTRINSIC OR MCP BINDING ONLY

- The INITIALIZE statement is legal only for intrinsic or MCP binding.

THE INTERNAL BINDER ARRAY, CRIT_BLK_AC, HAS OVERFLOWED—THE BINDER CAPACITY HAS BEEN EXCEEDED

- Refer this problem to your Customer Service Representative.

THE LIBRARY ATTRIBUTES IN THE SUBPROGRAM DIFFER FROM THE HOST. THE HOST LIBRARY ATTRIBUTES WILL BE USED.

- The subprogram and the host program have a different number of attributes or else the attributes do not match. You need not have attributes in the subprogram because the attributes of the host program are always used.

THE MATCHING LIBRARY <NAME> COULD NOT BE FOUND IN THE HOST

- Binder could not find the named library referenced by the library object.

THE MERGE OF THE TARGET LEVELS HAS RESULTED IN A CODE FILE THAT CANNOT BE RUN ON ANY MACHINE

- Either the host program or previously bound subprograms have machine features not available for the target level of this subprogram, or this subprogram has machine features not available in the host program or previously bound subprograms.

THE NUMBER OF ARRAY DIMENSIONS IN THE SUBPROGRAM DIFFERS FROM THE NUMBER OF DIMENSIONS IN THE HOST

- When an array is shared as a global item between two programs, it must be declared with the same number of dimensions in both programs.

THE NUMBER OF INTERNAL BINDER FILES REQUIRED IS TOO GREAT - THE BINDER'S CAPACITY HAS BEEN EXCEEDED

- The number of file declarations reserved by Binder for subprogram files has been exceeded. Each time Binder regresses to a previous level to bind a nested external subprogram, an additional subprogram file declaration is required. In addition, each library or multiprocedure file opened by Binder is left open until all subprograms have been bound from it. Thus, if the number of library files is greater than the number of file declarations, this message results.
- Refer this problem to your Customer Service Representative.

THE NUMBER OF PARAMETERS IN THE SUBPROGRAM DIFFERS FROM THE NUMBER OF PARAMETERS IN THE HOST

- For binding to occur, you must declare the same number of parameters in both the host program and the subprogram to be bound.

THE OFFSET OF xxxx CANNOT BE REACHED FROM LEXICAL LEVEL xx

- As the execution lex level of a subprogram increases, the offset that can be specified in a VALC or NAMC operator decreases. If a program or subprogram at a low lex level declares many variables, it is possible that a subprogram at a higher lex level will not be able to reference all of them.

  An offset of 4096 and a lex level of 2 or above specified in the warning message indicates that you have exceeded the maximum number of global stack cells that can be referenced by the program. The maximum offset that can be referenced at lex level 2 is 4095.

  An offset of 8192 and a lex level of 1 indicates that the bound program has exceeded the maximum number of segment dictionary entries that is allowed.

- If the offset in the error message is 4096, you can reduce the number of global stack cells referenced by your program by limiting the number of variables declared as GLOBAL or OWN to only those that must have a global address.

  If the offset in the error message is 8192, you can try one of the following methods to avoid the error:

  – Modify your bound program to decrease the number of segment dictionary entries.

  – Bind all the subprograms into the unbound version of the host.

  – If the host is ALGOL and the SEGDESCABOVE compiler option is used, change the option to a lower number and recompile the host to allow more code segment entries to fit in the segment dictionary.

    If the preceding methods do not resolve the error, then change the structure of your program to use fewer code or data segments. You can move some of the subprograms into a library to reduce the size of the bound program.

THE PROCEDURE THAT IS BEING PASSED AS A PARAMETER WAS NOT DECLARED IN A FORMAL DECLARATION

- When the parameters expected by a formal procedure are specified in a formal declaration by both the program unit passing the procedure as an argument and the subprogram receiving the procedure as a parameter, then no parameter checking for the formal procedure is performed at execution time. This error results when either the receiver or caller specifies the procedure formally, but the program unit passing or receiving the formal procedure does not contain a formal declaration of the procedure.

THE RESERVED WORD "FOR" WAS EXPECTED HERE

- In a USE statement, the word *FOR* after the first identifier is missing.

THE RESERVED WORD "FROM" WAS EXPECTED HERE

- In a BIND statement that begins with *BIND* =, the word *FROM* is missing after the equal sign (=).

THE RESERVED WORD "IS" WAS EXPECTED HERE

- In a HOST statement, the word *IS* is missing after the word *HOST*.

THE RESULTING CODE FILE WILL RUN ON A MORE RESTRICTED SET OF MACHINES THAN THE HOST

- This warning message is given when a subprogram must run on a more restricted set of computers than the host program. The resulting bound code file will run only on the more restricted set of computers.

THE SDF FORM LIBRARY APPLICATION RECORD DESCRIPTION IN THE HOST DID NOT MATCH THE SUBPROGRAM.

- SDF form record libraries in the host program and the subprogram are different versions, although they have identical names. This might indicate that one or more forms in the form library were altered between compilation of the host program and the subprogram.
- Recompile the host program and the subprogram.

THE SUBPROGRAM IDENTIFIER CONTAINED TOO MANY QUALIFIERS

- The subprogram identifier contains more qualifiers (*OF <identifier>* clauses) than are legal.

THERE ARE TOO MANY ADDRESSED PROCEDURES

- The number of addressed C functions exceeds the limits of Binder.

THERE ARE TOO MANY CALLS TO FORMAL OR ADDRESSED PROCEDURES

- The number of calls to C functions through pointers exceeds the limits of Binder.

THERE HAS BEEN A COMPILER ERROR. THE COMPILER EMITTED A BRANCH OPERATOR WITH AN OFFSET THAT IS TOO LARGE FOR THE LEXICAL LEVEL D2

- Refer this error to your Customer Service Representative.

THERE HAS BEEN A COMPILER ERROR. THE COMPILER EMITTED TOO MANY BRANCHES IN THE LEXICAL LEVEL D2

- Refer this error to your Customer Service Representative.

THERE IS A MISMATCH IN THE PARAMETER TYPE. IT IS BEING PASSED BY-NAME AND SHOULD BE PASSED BY-VALUE OR VICE VERSA

- This message is given in the following situations:
  - During the binding of an exported procedure, the host program declares a parameter by value, and the subprogram declares it by name, or vice versa.
  - During ALGOL-ALGOL, COBOL-COBOL, or ALGOL-COBOL bindi{ng, the host program declares a parameter by value, and the subprogram declares it by name, or vice versa.

THERE IS A MISMATCH IN THE ROW SIZE OF THE HEAP (POSSIBLY CAUSED BY DIFFERENT LONGLIMITS OR DIFFERENT MEMORY MODELS): THE SIZE IN THE HOST = nnnn, THE SIZE IN THE SUBPROGRAM = nnnn.

- The value of the LONGLIMIT compiler control option must be the same for a C host program and a C subprogram when: (1) binding object code files compiled with a C compiler before SSR 42.3, or (2) the compiler control option FARHEAP is false.

THERE IS A MISMATCH IN THE SIM CLASS INFORMATION. THE INTERNAL VALUE IN THE HOST = <VALUE>. THE INTERNAL VALUE IN THE SUBPROGRAM = <VALUE>.

- The CLASSINFOs of the SIM entity reference variable, entity reference array, or query variable used in the host program are different from those used in the subprogram. This error occurs when items with the same name are declared with different field sizes.

THERE IS A MISMATCH WITH THE (SIM) SEMANTIC QUALIFICATION. THE NAME IN THE HOST = <NAME>. THE NAME IN SUBPROGRAM = <NAME>.

- The database ID used to qualify the SIM entity reference variable, entity reference array, or query variable for the host program is different from the ID used for the subprogram. This error message can result if you tried to bind a host program and a subprogram compiled with different SIM databases.

THERE IS AN INCOMPATIBILITY IN THE ARRAY LOWER BOUNDS SPECIFICATION

- This error message results when Binder detects that a subprogram expects lower bounds for an array and they are not passed, or a subprogram does not expect lower bounds and is called with lower bounds passed to it.

  FORTRAN and FORTRAN77 always pass an array descriptor and an offset.

  COBOL rarely passes an offset, although it accepts and passes an offset if the *WITH LOWER BOUNDS* clause is used. However, the offset value itself is ignored in calculating subscripts within the COBOL subprogram.

  In ALGOL, the user can specify whether the array parameter is passed with lower bounds.

THERE IS AN INCORRECT NUMBER OF ADDRESS COUPLES DECLARED FOR THIS VARIABLE

- A PL/I structure of another variable type has a number of address couples associated with it, in accordance with the way it is declared in the program unit. This error occurs when two program units reference the same variable with a different number of address couples, indicating an incompatibility in the declarations within the separate program units.

THERE WERE NO SUBPROGRAMS BOUND TO THE HOST

- No subprograms were bound to the host program during the binding process.

THIS BINDER OPTION IS NOW OBSOLETE AND WILL BE IGNORED

- The Binder option you tried to use is obsolete.

THIS CODE FILE IS THE RESULT OF A PREVIOUS BIND AND IS SUITABLE AS A HOST ONLY

- The resultant code file from a previous bind cannot be bound to another host. Such a file may be used only as a host program in subsequent binds.

THIS CODE FILE USES AN UNRECOGNIZABLE MEMORY MODEL

- Refer this error to your Customer Service Representative.

THIS FILE CANNOT BE ACCESSED BY THE BINDER

- Binder is unable to access this file.

THIS FILE IS NOT A CODE FILE

- Check to make sure that the file title is complete and that a directory is not specified by the title.

THIS ITEM DEFINITION HAS ALREADY BEEN SEEN. ONLY ONE NON-EXTERNAL DECLARATION IS ALLOWED.

- The same variable or function is exported from more than one C subprogram or C host program.

THIS ITEM IS A COPY OF TWO OR MORE DIFFERENT ITEMS

- Refer this error to your Customer Service Representative.

THIS ITEM WAS INITIALIZED IN TWO OR MORE DECLARATIONS

- A C variable, array, or structure can be initialized only once. Binder has found an initialization in more than one C subprogram or C host program.

THIS NEW GLOBAL VARIABLE HAS BEEN ADDED TO THE HOST

- This is a warning message that indicates that a variable referenced in the subprogram does not exist in the host. Binder adds the variable to the host program at the global level.

THIS NUMBER IS TOO LARGE

- In an intrinsic number pair, either the first integer has a value greater than the largest possible installation number or the second integer has a value greater than the largest possible intrinsic number.

THIS IS A MISSPELLED CONTROL OPTION

- The specified compiler control option is not a valid Binder compiler control option. Binder recognizes only a specific set of compiler control options. Binder does not recognize user options.

THIS IS AN ILLEGAL <FAMILY NAME>

- The family name in a file specifier or directory specifier contains invalid characters. Valid characters are A through Z and 0 (zero) through 9.

THIS IS AN ILLEGAL IDENTIFIER

- This error is given in the following situations:
  - In a Binder control record, the item following the dollar sign ($) is not of identifier format.
  - In a BIND statement, an item following *OF* in a subprogram identifier is not an identifier.
  - In an EXTERNAL statement, either the item at the beginning of a subprogram identifier or an item following *OF* in a subprogram identifier is not an identifier.
  - In an INITIALIZE statement, the item following *INITIALIZE* or following a comma (,) is not an identifier.
  - In a USE statement, the item following *USE* or *FOR* or the item following *OF* in the subprogram identifier is not an identifier.

THIS IS AN INCORRECT INTRINSIC NUMBER BECAUSE ANOTHER INTRINSIC HAS THE SAME NUMBER

- Two intrinsics within the same intrinsic file cannot have the same intrinsic number pair.

THIS IS AN INCORRECT INTRINSIC NUMBER BECAUSE THE SAME INTRINSIC IDENTIFIER ALREADY EXISTS WITH A DIFFERENT NUMBER

- An intrinsic with the same identifier already exists within the intrinsic file. The existing intrinsic has an intrinsic number pair different from that specified for the intrinsic being bound.

THIS IS AN INVALID DATA DICTIONARY INVOCATION OR USAGE LIST

- Refer this problem to your Customer Service Representative.

THIS IS NOT A VALID BINDER STATEMENT

- The input to Binder is not one of the valid Binder statements.

THIS OBJECT CODE FILE HAS NO BINDER INFORMATION

- The file cannot be used for binding, either as a host program or as a subprogram. The option NOBINDINFO may have been set during the files creation, or the respective compiler may have determined that the file contained no external references, so it did not require binding.

THIS PROCEDURE CANNOT BE PASSED BETWEEN THESE TWO LANGUAGES

- A procedure cannot be passed as a parameter from the language in which the procedure call is written to the language in which the procedure declaration is written.

THIS PROCEDURE WAS NOT FOUND IN THE MULTIPROCEDURE FILE(S)

- Binder was unable to find the subprogram within the library files designated in the BIND statement. The subprogram is left in the host program in its present form, and binding of other subprograms continues.

THIS PROGRAM CONTAINS UNRECOGNIZED BINDINFO, POSSIBLY FOR A NEW CAPABILITY NOT SUPPORTED BY THIS BINDER

- One of the object files that you are attempting to bind was compiled with a compiler that has a later release level than the Binder program.
- Try the bind again using a Binder with a release level at least as great as the release level of the compiler.

THIS PROGRAM UNIT WAS COMPILED AT A LEXICAL LEVEL INCOMPATIBLE WITH THE LEXICAL LEVEL IN THE HOST. RECOMPILE USING THE OPTION $ SET LEVEL N

- The given subprogram was compiled at a lex level incompatible with its execution lex level within the host.
- Recompile the subprogram with the correct execution lex level by using the LEVEL option of the compiler.

THIS PROGRAM UNIT IS SUITABLE AS A HOST PROGRAM ONLY, SO IT CANNOT BE BOUND INTO ANOTHER HOST

- The designated subprogram file is actually a host program or main program. You cannot bind a host program to another host program.

THIS VARIABLE TYPE CANNOT BE ADDED TO THE HOST

- A variable referenced in a subprogram does not exist in the host. Usually Binder adds the variable to the host program. However, Binder is incapable of adding the following types of variables to a host:

  DATA BASE
  FORMAT

LABEL
LIBRARY
LIST
PICTURE
SDF form record libraries
STRING
Switch-type items
TRANSACTION BASE
TRANSLATETABLE
TRUTHSET
VALUE ARRAY

TO ADD A NEW LIBRARY OBJECT, LIBRARY <NAME> MUST BE COMPILED WITH A MARK 3.8 OR LATER COMPILER

- To add a new library object, the host program must be compiled with a Mark 3.8 or later version of the compiler.

TOO MANY ENTRIES—INCREASE MAXENTRIES

- Refer this problem to your Customer Service Representative.

WARNING: IF DATABASE WAS REORGANIZED AFTER THIS SUBPROGRAM WAS COMPILED, RECOMPILE TO PREVENT POSSIBLE DATA CORRUPTION

- This warning occurs when the host or a subprogram is compiled with an earlier version of the compiler that does not emit the database update timestamp. Without this timestamp, Binder cannot verify that the host and subprograms were compiled against the same version of the database. To prevent run-time data corruption, recompile the host or subprogram with a later version of the compiler that emits the timestamp, so that Binder can verify the database for you. Otherwise, you must verify the following for each subprogram that receives this message:
  - You have not reorganized the database since the host program or subprogram was compiled.
  - The host program or subprogram does not access any database structures that were changed in the database.

  If you cannot verify the items listed above, you must recompile the host program or the subprogram and rebind to ensure that the bound code file does not corrupt the database.

Appendix B
Using Binder Control Record Options

You can control the manner in which Binder processes the subprogram file and the host program file by including Binder control records in the WFL job or CANDE file used to execute Binder. In each Binder control record, you include one or more Binder options. These options allow you to

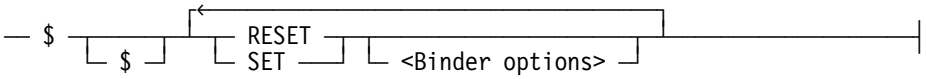
- Determine the content of printed output
- Determine whether error messages get sent to the ERRORS file and get printed
- Indicate whether a host file is required
- Determine whether lineinfo and bindinfo are included in the code file
- Determine whether a bound subprogram array is resized to match the host program array
- Enable intrinsic binding
- Prevent the code file from being locked when Binder cannot locate a subprogram
- Temporarily suspend binding when a subprogram is not available

Binder Control Record Format

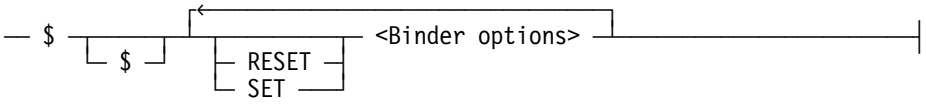
A Binder control record is identified by a dollar sign ($) in column one; a blank in column one and a dollar sign in column two; or dollar signs in columns one and two. Binder options follow the dollar sign in the succeeding columns through column 72. A percent sign (%) appearing in any column from 2 through 72 of a Binder control record indicates that the remaining columns of the record are to be ignored by the Binder. Binder control records can occur at any point in the Binder input file.

There are two formats for including options in Binder control records. Syntax 1 allows you to specify options that are effective throughout the binding process. Syntax 2 allows you to assign the value TRUE to certain options for the duration of the binding of specific subprograms.

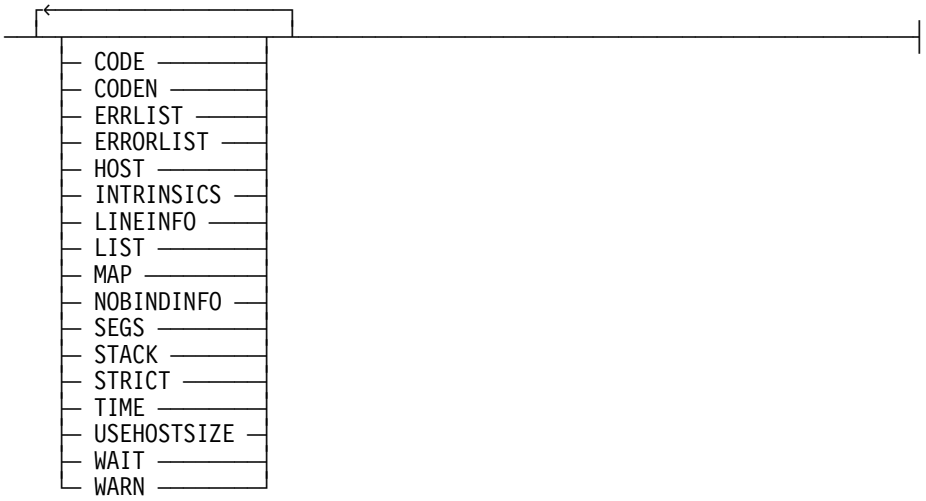
Syntax 1, Version A



Syntax 1, Version B



<Binder options>



Explanation

Syntax 1 lets you specify Binder options that are effective throughout the binding process. Syntax 1 has two versions, A and B.

Version A

The Binder control record contains either one or two dollar signs ($) followed by either *SET* or *RESET*, followed by one or more Binder options. If the action is *SET*, the named options are assigned a value of TRUE. If the action is *RESET*, the named options are assigned a value of FALSE.

If you do not name any options, all options are set or reset according to the action you specify.

Example

```
$ SET CODE STACK LIST
$ RESET SEGS
```

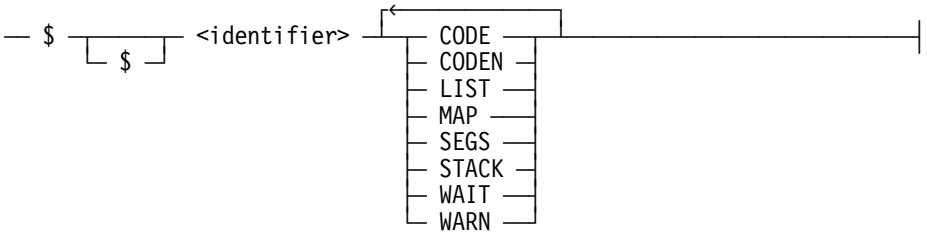
Version B

The Binder control record contains either one or two dollar signs ($) followed by one or more Binder options. *SET* and *RESET* are not used to indicate values of TRUE or FALSE. Rather, the named options are assigned a value of TRUE, and the unnamed options are assigned a value of FALSE. If the control record contains a dollar sign and no options, Binder ignores the record. (Note that ERRORLIST (ERRLIST), LINEINFO, and STRICT cannot be used in this syntax, so they assume their default values.)

Example

```
$ CODE STACK LIST
```

Syntax 2



Explanation

Syntax 2 lets you set certain options to TRUE for the binding of a specified subprogram. All named options are assigned a value of TRUE. All unnamed options are assigned a value of FALSE.

The options assume the assigned values only during the binding of the subprogram specified by the identifier. Once the subprogram is bound, all options are restored to their previous values. For any option, the last setting in a control record of Syntax 1 takes precedence over all other settings.

You can include only one subprogram identifier in each Binder control record. You must include one or more Binder options after the identifier.

Example

```
$ PROCEDURE1 CODE WAIT WARN
```

For information about identifiers, see Section 2.

Binder Options

Binder options are discussed in alphabetical order in Table B–1:

Table B–1. Binder Options

Option	Value	Function
CODE	False	Indicates whether the resultant code file will be printed in hexadecimal form
CODEN	False	Indicates whether the input code files will be printed in hexadecimal form
ERRORLIST	False (True for binds initiated by CANDE)	Indicates whether Binder will write error messages to the file titled, ERRORS. If Binder is initiated from WFL, ERRORS is a printer file. If Binder is initiated from CANDE, ERRORS is a remote file, and messages are written to the remote station that initiated the bind. (ERRORLIST is the preferred synonym for ERRLIST.)
ERRLIST	False (True for binds initiated by CANDE)	See the preferred synonym, ERRORLIST.
HOST	False	Indicates whether a host file is required. When the INTRINSICS options is FALSE, a host file is always required, and the HOST option has no effect. When the HOST option is TRUE and the INTRINSICS option is TRUE, a host file is required and is used for intrinsic binding. When the HOST option is FALSE and the INTRINSICS options is TRUE, a host file is not required and is not used.
INTRINSICS	False	Indicates whether an intrinsic file will be created or intrinsic binding will be enabled. When FALSE, the INTRINSICS option can still create an intrinsic code file if the host file is the object file of a previous intrinsic bind. If the INTRINSICS option is FALSE, a host file is always required and the HOST option has no effect.
LINEINFO	True	Indicates whether the resulting code file will contain all LINEINFO encountered in the host and subprogram files.
LIST	True (False for binds initiated by CANDE)	Indicates whether input records, identifiers and their address couples, and BEGIN BINDING and END BINDING messages will be printed.

continued

Table B–1. Binder Options (cont.)

Option	Value	Function
MAP	False	Indicates whether the address couples of all identifiers in the resultant code will be printed, both in alphabetical order by identifier and in address couple order. (The MAP option is the preferred synonym for the STACK option.)
NOBINDINFO	False	Indicates whether the Binder will purge all Binder information from the resultant code file. The resultant code file cannot then be used as a host for a subsequent bind if the value of NOBINDINFO is TRUE.
SEGS	True (False for binds initiated by CANDE)	Indicates whether the segment dictionary changes will be printed. Assigning a value to the LIST option causes the same value to be assigned to the SEGS option.
STACK	False	See the preferred synonym, MAP.
STRICT	False (True for MCP binds)	Indicates whether the resultant code file will be locked if a subprogram specified in a BIND statement is not bound. When FALSE, the code file is locked.
TIME	False	Indicates whether header and trailer information for the bind will be printed. Because the information is printed when LIST is TRUE, the value of TIME is significant only when LIST is FALSE.
USEHOSTSIZE	False	Indicates whether an array global to a bound subprogram is resized to the size of its corresponding array in the host. If the USEHOSTSIZE option is not set (FALSE), the larger array size from either the host or the subprogram is used.
WAIT	False	Indicates whether Binder will suspend the binding process if a specified subprogram file is not present. Upon suspension, the operator is allowed to make the file present, to terminate the Binder, or to enter the OF (Optional File) or FA (File Attribute) ODT command. Any OF or FA command applies to that file only. Subsequent nonpresent files again cause the Binder to suspend binding. For more information about the OF and FA commands, refer to the *System Commands Reference Manual*.

continued

Table B–1. Binder Options (cont.)

Option	Value	Function
WARN	True (False for binds initiated by CANDE)	Indicates whether warning messages will be printed upon the occurrence of certain conditions. When WARN is FALSE, these warning messages are suppressed. Assigning a value to the LIST option causes the same value to be assigned to the WARN option.

Appendix C
Understanding Railroad Diagrams

This appendix explains railroad diagrams, including the following concepts:

- Paths of a railroad diagram
- Constants and variables
- Constraints

The text describes the elements of the diagrams and provides examples.

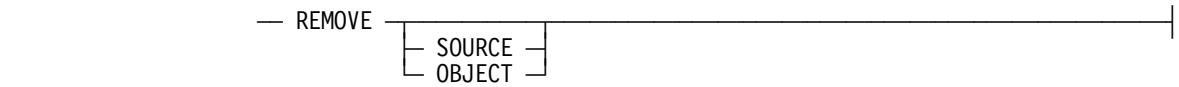
Railroad Diagram Concepts

Railroad diagrams are diagrams that show you the standards for combining words and symbols into commands and statements. These diagrams consist of a series of paths that show the allowable structures of the command or statement.

Paths

Paths show the order in which the command or statement is constructed and are represented by horizontal and vertical lines. Many commands and statements have a number of options so the railroad diagram has a number of different paths you can take.

The following example has three paths:



The three paths in the previous example show the following three possible commands:

- REMOVE
- REMOVE SOURCE
- REMOVE OBJECT

A railroad diagram is as complex as a command or statement requires. Regardless of the level of complexity, all railroad diagrams are visual representations of commands and statements.

Railroad diagrams are intended to show

- Mandatory items
- User-selected items
- Order in which the items must appear
- Number of times an item can be repeated
- Necessary punctuation

Follow the railroad diagrams to understand the correct syntax for commands and statements. The diagrams serve as quick references to the commands and statements.

The following table introduces the elements of a railroad diagram:

Table C–1. Elements of a Railroad Diagram

The diagram element...	Indicates an item that...
Constant	Must be entered in full or as a specific abbreviation
Variable	Represents data
Constraint	Controls progression through the diagram path

Constants and Variables

A constant is an item that must be entered as it appears in the diagram, either in full or as an allowable abbreviation. If a constant is partially boldfaced, you can abbreviate the constant by

- Entering only the boldfaced letters
- Entering the boldfaced letters plus any of the remaining letters

If no part of the constant is boldfaced, the constant cannot be abbreviated.

Constants are never enclosed in angle brackets (< >) and are in uppercase letters.

A variable is an item that represents data. You can replace the variable with data that meets the requirements of the particular command or statement. When replacing a variable with data, you must follow the rules defined for the particular command or statement.

In railroad diagrams, variables are enclosed in angle brackets.

In the following example, BEGIN and END are constants, whereas <statement list> is a variable. The constant BEGIN can be abbreviated since it is partially boldfaced.

Valid abbreviations for BEGIN are

- BE
- BEG
- BEGI

Constraints

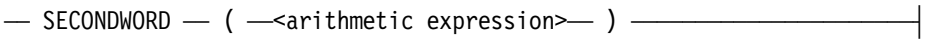
Constraints are used in a railroad diagram to control progression through the diagram. Constraints consist of symbols and unique railroad diagram line paths. They include

- Vertical bars
- Percent signs
- Right arrows
- Required items
- User-selected items
- Loops
- Bridges

A description of each item follows.

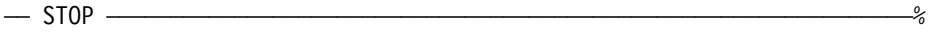
Vertical Bar

The vertical bar symbol (|) represents the end of a railroad diagram and indicates the command or statement can be followed by another command or statement.



Percent Sign

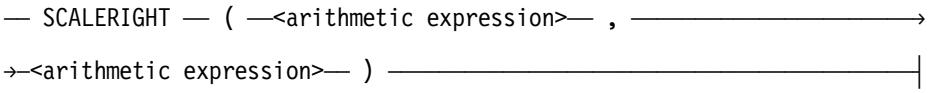
The percent sign (%) represents the end of a railroad diagram and indicates the command or statement must be on a line by itself.



Right Arrow

The right arrow symbol (>)

- Is used when the railroad diagram is too long to fit on one line and must continue on the next
- Appears at the end of the first line, and again at the beginning of the next line



Required Item

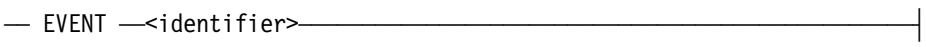
A required item can be

- A constant
- A variable
- Punctuation

If the path you are following contains a required item, you must enter the item in the command or statement; the required item cannot be omitted.

A required item appears on a horizontal line as a single entry or with other items. Required items can also exist on horizontal lines within alternate paths, or nested (lower-level) diagrams.

In the following example, the word EVENT is a required constant and <identifier> is a required variable:



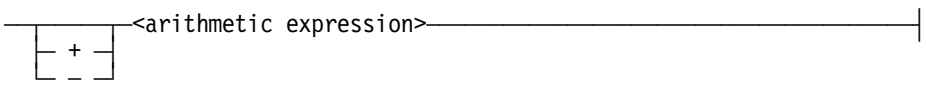
User-Selected Item

A user-selected item can be

- A constant
- A variable
- Punctuation

User-selected items appear one below the other in a vertical list. You can choose any one of the items from the list. If the list also contains an empty path (solid line) above the other items, none of the choices are required.

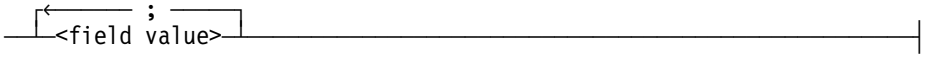
In the following railroad diagram, either the plus sign (+) or the minus sign (–) can be entered before the required variable <arithmetic expression>, or the symbols can be disregarded because the diagram also contains an empty path.



Loop

A loop represents an item or a group of items that you can repeat. A loop can span all or part of a railroad diagram. It always consists of at least two horizontal lines, one below the other, connected on both sides by vertical lines. The top line is a right-to-left path that contains information about repeating the loop.

Some loops include a return character. A return character is a character—often a comma (,) or semicolon (;)—that is required before each repetition of a loop. If no return character is included, the items must be separated by one or more spaces.



Bridge

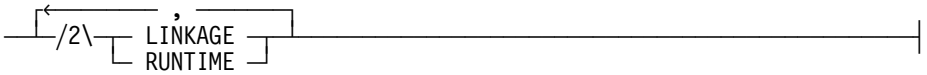
A loop can also include a bridge. A bridge is an integer enclosed in sloping lines (/ \) that

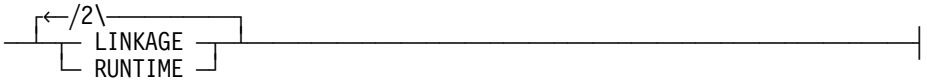
- Shows the maximum number of times the loop can be repeated
- Indicates the number of times you can cross that point in the diagram

The bridge can precede both the contents of the loop and the return character (if any) on the upper line of the loop.

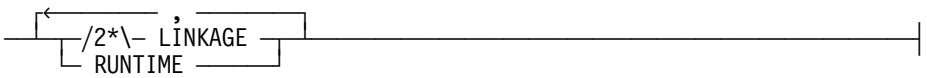
Not all loops have bridges. Those that do not can be repeated any number of times until all valid entries have been used.

In the first bridge example, you can enter LINKAGE or RUNTIME no more than two times. In the second bridge example, you can enter LINKAGE or RUNTIME no more than three times.





In some bridges an asterisk (*) follows the number. The asterisk means that you must cross that point in the diagram at least once. The maximum number of times that you can cross that point is indicated by the number in the bridge.



In the previous bridge example, you must enter LINKAGE at least once but no more than twice, and you can enter RUNTIME any number of times.

Following the Paths of a Railroad Diagram

The paths of a railroad diagram lead you through the command or statement from beginning to end. Some railroad diagrams have only one path; others have several alternate paths that provide choices in the commands or statements.

The following railroad diagram indicates only one path that requires the constant LINKAGE and the variable <linkage mnemonic>:

Alternate paths are provided by

- Loops
- User-selected items
- A combination of loops and user-selected items

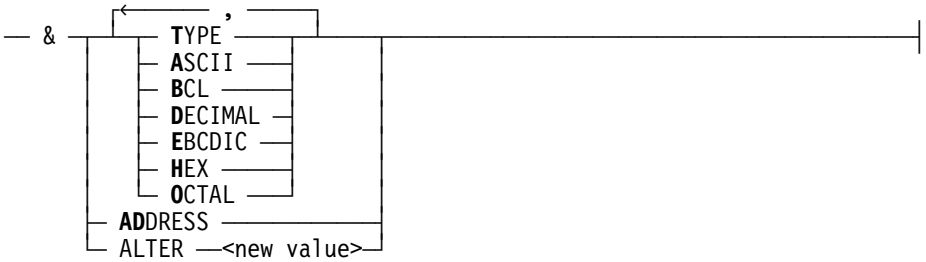
More complex railroad diagrams can consist of many alternate paths, or nested (lower-level) diagrams, that show a further level of detail.

For example, the following railroad diagram consists of a top path and two alternate paths. The top path includes

- An ampersand (&)
- Constants that are user-selected items

  These constants are within a loop that can be repeated any number of times until all options have been selected.

The first alternative path requires the ampersand and the required constant ADDRESS. The second alternative path requires the ampersand followed by the required constant ALTER and the required variable <new value>.

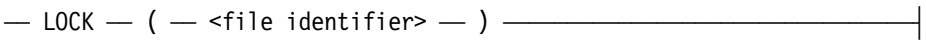


Railroad Diagram Examples with Sample Input

The following examples show five railroad diagrams and possible command and statement constructions based on the paths of these diagrams.

Example 1

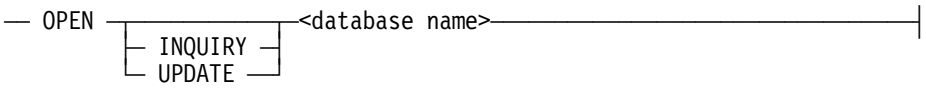
<lock statement>



Sample Input	Explanation
LOCK (FILE4)	LOCK is a constant and cannot be altered. Because no part of the word is boldfaced, the entire word must be entered. The parentheses are required punctuation, and FILE4 is a sample file identifier.

Example 2

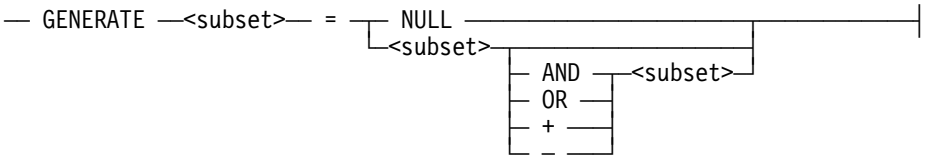
<open statement>



Sample Input	Explanation
OPEN DATABASE1	The constant OPEN is followed by the variable DATABASE1, which is a database name. The railroad diagram shows two user-selected items, INQUIRY and UPDATE. However, because an empty path (solid line) is included, these entries are not required.
OPEN INQUIRY DATABASE1	The constant OPEN is followed by the user-selected constant INQUIRY and the variable DATABASE1.
OPEN UPDATE DATABASE1	The constant OPEN is followed by the user-selected constant UPDATE and the variable DATABASE1.

Example 3

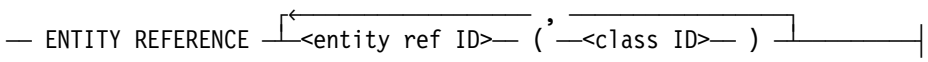
<generate statement>



Sample Input	Explanation
GENERATE Z = NULL	The GENERATE constant is followed by the variable Z, an equal sign (=), and the user-selected constant NULL.
GENERATE Z = X	The GENERATE constant is followed by the variable Z, an equal sign, and the user-selected variable X.
GENERATE Z = X AND B	The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the AND command (from the list of user-selected items in the nested path), and a third variable, B.
GENERATE Z = X + B	The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the plus sign (from the list of user-selected items in the nested path), and a third variable, B.

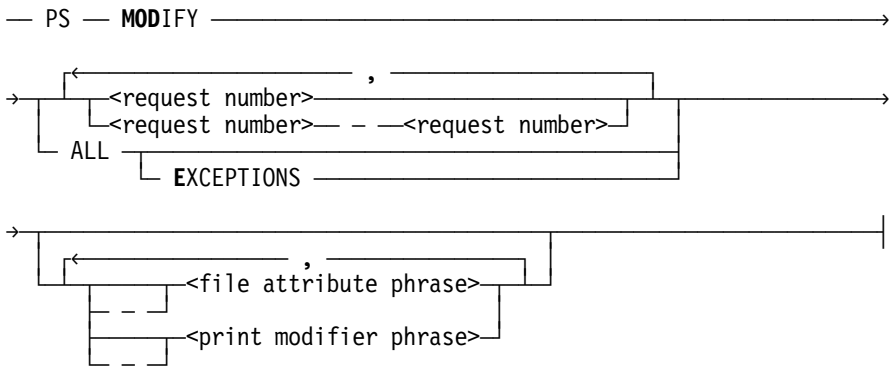
Example 4

<entity reference declaration>



Sample Input	Explanation
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR)	The required item ENTITY REFERENCE is followed by the variable ADVISOR1 and the variable INSTRUCTOR. The parentheses are required.
ENTITY REFERENCE ADVISOR1 (INSTRUCTOR), ADVISOR2 (ASST_INSTRUCTOR)	Because the diagram contains a loop, the pair of variables can be repeated any number of times.

Example 5



Sample Input	Explanation
PS MODIFY 11159	The constants PS and MODIFY are followed by the variable 11159, which is a request number.
PS MODIFY 11159,11160,11163	Because the diagram contains a loop, the variable 11159 can be followed by a comma, the variable 11160, another comma, and the final variable 11163.
PS MOD 11159–11161 DESTINATION = "LP7"	The constants PS and MODIFY are followed by the user-selected variables 11159–11161, which are request numbers, and the user-selected variable DESTINATION = “LP7”, which is a file attribute phrase. Note that the constant MODIFY has been abbreviated to its minimum allowable form.
PS MOD ALL EXCEPTIONS	The constants PS and MODIFY are followed by the user-selected constants ALL and EXCEPTIONS.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Special Characters