Rdok 5249 → BB 150

Burroughs

RELATIVE TO
~~III.0~~
III.1

B 7000 / B 6000

BINDER

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A

5-77

5001456

5249

Burroughs

B 7000/B 6000

BINDER

REFERENCE MANUAL

PRICED ITEM

Printed in U. S. America

5-77

5001456

LIST OF EFFECTIVE PAGES

Page No.	Issue
Title	Original
A	Original
i thru v	Original
vi blank	Original
1-1	Original
1-2 blank	Original
2-1 thru 2-4	Original
3-1 thru 3-2	Original
4-1 thru 4-4	Original
5-1 thru 5-8	Original
6-1 thru 6-11	Original
6-12 blank	Original
7-1 thru 7-10	Original
8-1 thru 8-2	Original
A-1	Original
A-2	Original
B-1 thru B-5	Original
B-6 blank	Original
C-1 thru C-2	Original
D-1 thru D-8	Original
Index-1 thru Index-4	Original

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001456-001 Date: November, 1979

Publication Title: B 7000/B 6000 BINDER Reference Manual

Other Affected Publications: None

Supersedes: N/A

Description:

Revisions to the text are indicated by a black vertical bar on the affected pages.

Replace these pages	Add page
iii	7-2A
v	
2-1	
5-3	
6-7	
6-9	
7-1	
7-3	
7-5	
A-1	
B-5	
Index-1	
Index-3	

Retain this PCN as a record of changes made to the basic publication.

Printed in U. S. America

For Library Binder

PCN 5001456-001

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue
i thru iii	Original	7-2A	PCN-001
iv thru v	PCN-001	7-2B	Blank
vi	Blank	7-3	Original
1-1	Original	7-4	PCN-001
1-2	Blank	7-5	Original
2-1	PCN-001	7-6	PCN-001
2-2 thru 2-4	Original	7-7 thru 7-10	Original
3-1 thru 3-2	Original	8-1 thru 8-2	Original
4-1 thru 4-4	Original	A-1	PCN-001
5-1 thru 5-3	Original	A-2	Blank
5-4	PCN-001	B-1 thru B-4	Original
5-5 thru 5-8	Original	B-5	PCN-001
6-1 thru 6-7	Original	B-6	Blank
6-8 thru 6-9	PCN-001	C-1 thru C-2	Original
6-10 thru 6-11	Original	D-1 thru D-8	Original
6-12	Blank	Index-1 thru Index-3	PCN-001
7-1 thru 7-2	PCN-001	Index-4	Original

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

Rdok 10379

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001456-002 Date: February, 1981

Publication Title: B 7000/B 6000 BINDER Reference Manual (May 1977)

Other Affected Publications: None

Supersedes: N/A

Description

Revisions to the text are indicated by a black vertical bar on the affected pages.

Replace these pages

i
2-1
4-3
5-3
6-1
7-1
7-2A
7-5
7-7
8-1
B-5

Retain this PCN as a record of changes made to the basic publication.

The above pages covering
PCN 5001456-002

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue
Title	Original	6-8 thru 6-9	PCN-001
A	PCN-002	6-10 thru 6-11	Original
B	PCN-002	6-12	Blank
i thru ii	PCN-002	7-1 thru 7-2	PCN-002
iii thru v	PCN-001	7-2A	PCN-002
vi	Blank	7-2B	Blank
1-1	Original	7-3 thru 7-4	PCN-001
1-2	Blank	7-5 thru 7-8	PCN-002
2-1 thru 2-2	PCN-002	7-9 thru 7-10	Original
2-3 thru 2-4	Original	8-1 thru 8-2	PCN-002
3-1 thru 3-2	Original	A-1	PCN-001
4-1 thru 4-2	Original	A-2	Blank
4-3 thru 4-4	PCN-002	B-1 thru B-4	Original
5-1 thru 5-2	Original	B-5	PCN-002
5-3 thru 5-4	PCN-002	B-6	Blank
5-5 thru 5-8	Original	C-1 thru C-2	Original
6-1 thru 6-2	PCN-002	D-1 thru D-8	Original
6-3 thru 6-7	Original	Index-1 thru Index-4	PCN-001

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

PREFACE

This document provides the programmer with a complete description of the Binder language as implemented on the Burroughs B 7000/B 6000 data processing systems.

This reference manual is divided into the following eight sections and four appendixes.

Section 1, **INTRODUCTION**, lists some of the advantages gained by the implementation and practical application of program binding.

Section 2, **BINDING PROCESS**, explains the overall binding process.

Section 3, **BINDER SYNTAX CONVENTIONS**, explains the syntactical notation used in defining the Binder language.

Section 4, **LANGUAGE COMPONENTS**, describes the elements that form the most primitive structures of the Binder language.

Section 5, **BINDER STATEMENTS**, describes the language elements (⟨*binder statement*⟩s) which cause specified actions to be performed.

Section 6, **INTRALANGUAGE BINDING**, describes the procedures and techniques required for intralanguage binding.

Section 7, **INTERLANGUAGE BINDING**, describes the procedures and techniques required for interlanguage binding.

Section 8, **INTRINSIC BINDING**, explains the binding procedures required to create and bind intrinsic files.

Appendix A, **RESERVED WORDS**, provides a list of "words" that have been set aside for a specific purpose within the Binder language.

Appendix B, **BINDER OPTIONS**, describes the compiler options available to the user.

Appendix C, **BINDER FILES**, describes the various compiler files.

Appendix D, **ERROR MESSAGES**, lists the various error messages along with their meanings.

The information in the following documents pertains to and supplements the material presented in this reference manual:

Title	Form No.
B 5000/B 6000/B 7000 ALGOL Language Reference Manual	5011760
B 7000/B 6000 COBOL Language Reference Manual	5001464

Title	Form No.
B 7000/B 6000 FORTRAN Reference Manual	5001506
B 7000/B 6000 System Software Operational Guide (Vol. I)	5011661
B 5000/B 6000/B 7000 System Software Operational Guide (Vol. II)	5011679
B 7000/B 6000 PL/I Language Information Manual	5001530
B 6700 Quick-Reference Handbook	5001225

CONTENTS

ILLUSTRATIONS

TABLES

1. INTRODUCTION

The purpose of this document is to provide information concerning program binding as implemented on the B 7000/B 6000 data processing systems. Specifically, this document provides the Binder syntax and the techniques required to bind separately (externally) compiled program units (subprograms) into an executable program.

This manual is user-oriented in that it is designed to provide the user with a source of reference information pertinent to the various aspects and conventions of the Binder, as well as to the binding process.

The Binder is designed primarily to function as an efficiency tool, saving not only computer time in recompiling but programmer time in rewriting as well. For example, when a large program requires change, only that portion of the program requiring change needs to be rewritten and recompiled. The Binder is then invoked to insert (replace) the revision into the program. By following this procedure it is not necessary to rewrite or recompile the entire program to effectively change or correct an existing program.

Practical application of program binding provides the user with the ability to enhance the efficiency of existing programs while requiring only a minimum expense of rewrite and recompile time.

2. BINDING PROCESS

The Binder is used to combine one or more externally compiled subprograms into one resultant program. The externally compiled subprograms to be bound must have been constructed by the ALGOL, COBOL, ESPOL, FORTRAN, or PL/I compiler. (The NEWP MCP Host can have ALGOL procedures bound into it. Statements concerning ALGOL in this manual also apply to its various extensions, such as DCALGOL and DMALGOL). Each of the respective compilers is capable of producing a complete and executable program with no unresolved external references, so long as all program units are compiled together in one compilation. However, if one or more subprograms are left external during the compilation, the Binder must be used to bind the external subprogram(s) into the initially compiled program which produces a "bound" program.

In order for the Binder to perform the binding process, which includes a number of optional listings in addition to the bound program, the Binder must be supplied with other types of input (card input files, Host files, and subprogram files) in addition to the externally compiled subprograms. Binder input and output files are illustrated in figure 2-1.

BINDER INPUT FILES

In a normal execution, the Binder is given three sources of input: a Host file, a card input file, and one or more externally compiled subprograms (discussed below). A Host file is always required except for intrinsic binding. (Refer to **INTRINSIC BINDING**, section 8.) Use of the card input file is optional. Use of subprogram files is also optional in the sense that the Binder will attempt to bind them into the Host only as they are required to satisfy an EXTERNAL reference within the Host or as directed by a ⟨*bind statement*⟩. (Refer to **BINDER STATEMENTS**, section 5.)

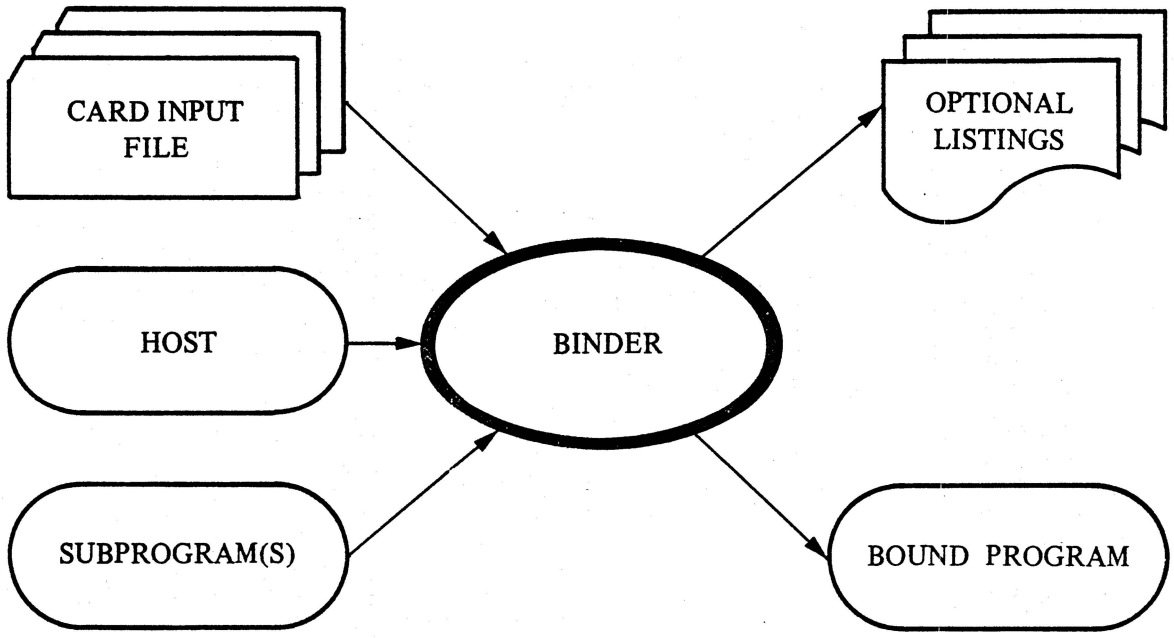


Figure 2-1. Binding Process

Host File

The program to which the separate subprograms are bound is referred to as the Host file. The title of the Host file may be specified to the Binder in one of two ways: by a ⟨*host statement*⟩, or by label equating the Host file in the Work Flow Language (WFL) syntax. A Host file may be the resultant code file of a previous bind, and it always contains the first executable code segment of the resultant bound program.

Examples of a Host file are as follows: an ALGOL outer block, a FORTRAN main program, the MCP, a PL/I procedure, a COBOL program compiled at level 2, or a previously bound program.

Card Input File

The card input file is used to set various options within the Binder, to indicate which subprograms to bind, and to specify the file names in which various subprograms are located.

Subprogram File

The title(s) of a subprogram file(s) is supplied to the Binder through the appropriate syntax of the ⟨*bind statement*⟩. A subprogram file may not be the resultant program of a previous bind. The following are examples of subprogram files: and ALGOL procedure, a FORTRAN subroutine or function, a separately compiled procedure of the MCP, an intrinsic, a PL/I procedure, or a COBOL program compiled with the LEVEL option set greater than 2.

BINDER EXECUTION

The Binder begins execution by attempting to read the card input file if one exists. If a card file does not exist, the Binder attempts to open the Host file as discussed below. The information within the card deck is processed and stored for future reference by the Binder. If any syntax errors are detected during the processing, the Binder terminates after reading the last input card.

The Binder then attempts to open the Host file, and unconditionally attempts to read the first record of the Host. If the Host file is not present or cannot be made present, the Binder must be terminated by the operator. If the Host file is not a code file or is otherwise not suitable for binding, an appropriate error message is displayed and the Binder terminates.

Once the Host file has been found and opened, the Binder locates and reads the binder information contained therein. In processing this information, the Binder examines each subprogram named to determine its current condition (bound or unbound) and whether a statement from the card input deck applies to it. Table 2-1 indicates the action taken by the Binder as a result of this examination.

Table 2-1. Binder Examination Result Action

STATEMENT	CONDITION OF NAMED SUBPROGRAM	
	BOUND	UNBOUND
No statement	Ignores subprogram.	Attempts to bind subprogram.
⟨*external statement*⟩	Ignores subprogram.	Ignores subprogram.
⟨*bind statement*⟩	Binds New Subprogram. and Discards "old."	Binds subprogram.

When an indication of an unbound subprogram is found, the Binder attempts to find the correct file from which the subprogram may be bound. If the correct file is not present on disk, and the WAIT option is not specified (see Binder Options, appendix B), the Binder ignores the unbound condition, emits an appropriate message, and continues to look for other subprograms to bind. If the file is present, it is checked to verify that it contains the information necessary for binding, that the file is not another Host file (one Host may not be bound to another), and that the file is not the resultant code file of a previous bind. If any of these conditions exist, the Binder prints an appropriate message and terminates binding the given subprogram as if the file had not been present on disk.

After the subprogram file has been opened, the description of the subprogram itself is checked first to verify that it 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, the Binder discontinues binding the subprogram, returns to its previous level of binding, and continues looking for other subprograms to bind. The subprogram for which a bind was attempted is treated as if it were not present on disk.

Once the subprogram description has been matched with the Host's description, any subsequent error messages arising during the bind of that subprogram will cause a fatal error condition, i.e., binding of that subprogram will terminate.

Binding error messages are produced when the Binder compares the description of a global reference made in the subprogram with its corresponding description in the Host file and finds a mismatch. Minor discrepancies on type matching are allowed, e.g., referencing a variable as a real in the subprogram when it is declared as an integer in the Host. But other mismatches are flagged as fatal errors, e.g., referencing a single-precision variable as an array, or calling another subprogram with the wrong number of parameters.

BINDER OUTPUT FILE

As illustrated in figure 2-1, two types of outputs are produced by the Binder during normal execution: a bound program and various printer listings.

Bound Program

The bound program produced by the Binder consists of the Host program and all subprograms bound into the Host. The bound program is usually complete in that it is normally capable of being executed. However, the bound program may still contain one or more unresolved external references. Recall that when a subprogram to be bound is not present on disk at binding time, and the WAIT option is not specified, the Binder prints an appropriate message and continues to look for other subprograms to bind. As a result of this procedure the specified subprogram is not bound and the external reference within the Host remains unresolved.

An unresolved external reference may or may not be fatal to the execution of the bound program. The fatal condition arises when the bound program, during execution, attempts to execute an external procedure. As a result of this attempt, an error message is displayed and the program is terminated. An unresolved external reference within the bound program is not a fatal condition unless an attempt to access the external subprogram is made. To prevent the occurence of this type of error, the WAIT option should be specified (see appendix B) and the Binder printer listing checked to verify that the binding of all necessary subprograms has been performed.

ALLOWABLE BINDING COMBINATIONS

The Binder is capable of binding programs and subprograms which have been constructed by the various compilers into one resultant executable program. However, not all of the various programming languages are capable of being bound together. Table 2-2 lists the languages which may be bound and also indicates the allowable binding combinations which may be performed. For additional information regarding the allowable intralanguage and interlanguage binding requirements, refer to sections 6 and 7, respectively.

Table 2-2. Allowable Binding Combinations

SEPARATE SUBPROGRAM	HOST LANGUAGE				
	ALGOL	COBOL	ESPOL	FORTRAN	PL/I
ALGOL	X	X	X	X	
COBOL	X	X		X	
ESPOL			X		
FORTRAN	X	X		X	
PL/I					X

REDUCING BINDING TIME

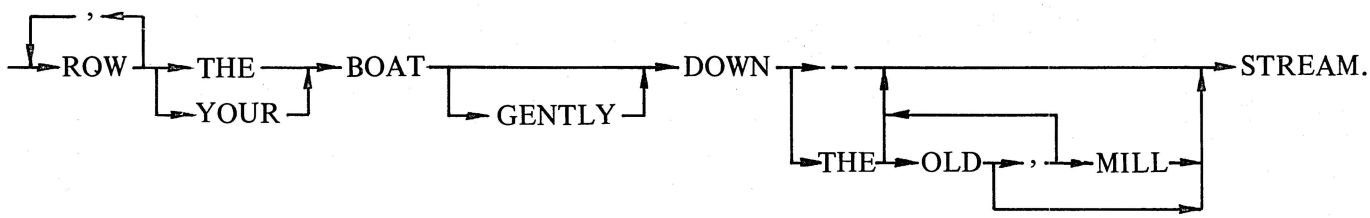
The major portion of the Binder time is taken up by input and output operations. By using the capabilities of rebinding to a bound Host and replacement binding of a given subprogram, one can substantially reduce the amount of time required for binding. Consider the case where one has a Host file plus 250 files containing subprograms to be bound to it. To update an existing subprogram, a user recompiles the separate subprogram and the existing file is replaced on disk by the new version. To bind the program 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. By using the LIBRARY option available in some compilers, one could reduce the number of files by compiling numerous subprograms into one library file. The most efficient utilization of the Binder, however, is to maintain a Host which contains a completely bound program and to update existing subprograms by recompiling the new version of the subprogram and replacing the existing subprogram in the Host by binding in the new version. This latter method, requiring only two files to be accessed, reduces the time by a factor of 10.

Object-Code Efficiency

In general, the code contained in a bound program is equivalent to code contained in similar files which are compiled and linked together by a compiler. In the case of replacement binding, the Binder reuses all segment dictionary locations which were used exclusively by the subprogram being replaced. Code segments which are shared among several subprograms are retained in the bound program until all subprograms using those code segment are replaced. D2 stack level items, once added to a bound program, are never removed. For example, if a subprogram containing OWN or STATIC variables is replaced, the D2 level locations for the OWN(s) declared in the replaced subprogram are not reused; rather new D2 locations are allocated for the subprogram replacing the existing one. In most cases the unreferenced D2 stack locations representing OWN(s) belonging to replaced subprograms cause extremely little overhead in execution or in core usage. However, if the OWN is an initialized array (an array containing initial values other than zero), the code to initialize the array will cause the array to be made present. Therefore, repeated replacement binding of a subprogram containing initialized arrays could cause some additional core usage.

3. BINDER SYNTAX CONVENTIONS

The syntax diagram is the method used to depict the Binder syntax. This method affords a very concise and lucid exposition of syntax; it is rigorous without being cumbersome. Any path traced along the forward directions of the arrows produces a syntactically valid statement. The following example illustrates the technique:



Valid constructs from this syntax diagram include:

ROW THE BOAT DOWN-STREAM.
ROW,ROW,ROW YOUR BOAT GENTLY DOWN THE STREAM.
ROW,ROW,ROW,ROW THE BOAT DOWN THE OLD STREAM.
ROW YOUR BOAT DOWN THE OLD,MILL STREAM.
ROW THE BOAT DOWN THE OLD,MILL STREAM.

KEY WORDS

Bold-face symbols and uppercase letters in syntax diagrams indicate symbols and words which appear literally in the construct.

SYNTACTIC VARIABLES

In the syntax diagrams, left and right broken brackets (⟨ ⟩) are used to contain syntactic variables that represent information to be supplied by the programmer.

The following is an example of a syntactic variable that appears in a syntax diagram.

BIND = FROM ⟨*file specifier*⟩;

Braces ({ }) are used to enclose syntactic variable expressions defined by the meaning of the English-language expression contained within the braces. For example, the following syntactic variable expression:

{one of the EBCDIC characters, 0 thru 2, inclusive}

would include one of the following characters:

0
1
2

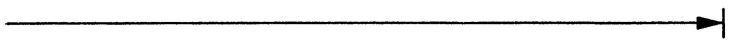
CONSTRUCT TERMINATOR

Constructs in the Binder language must be terminated by a semicolon (;) as illustrated below.

BIND = FROM ⟨*file specifier*⟩ ;

The semicolon is part of the syntax and must appear following the construct.

A vertical bar (|) is used to graphically illustrate the termination of a syntax diagram. This can be illustrated as follows:

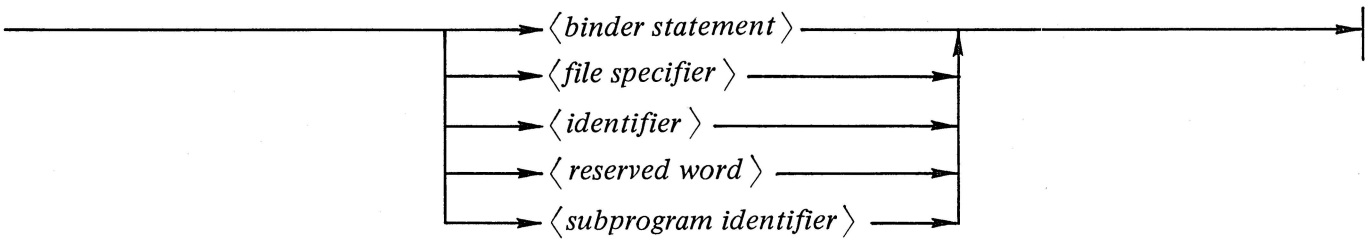


The vertical bar is not part of the syntax, but merely indicates the termination of a construct.

4. LANGUAGE COMPONENTS

LANGUAGE COMPONENTS

Syntax



Examples

A/B/C
SUBPROG

Semantics

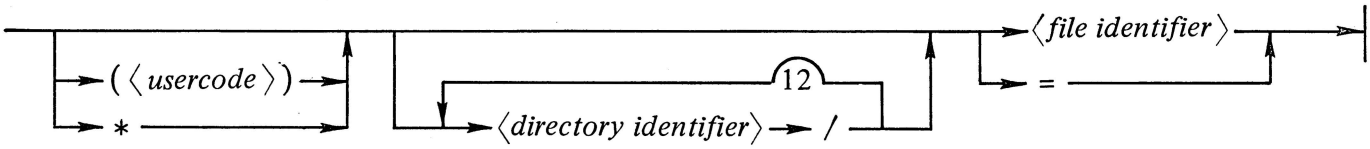
All ⟨*binder statement*⟩s are discussed in section 5.

A complete list of the ⟨*reserved word*⟩s is contained in appendix A.

All other ⟨*language component*⟩s are described in this section.

FILE SPECIFIER

Syntax



Examples

A/B/C
A/B/=
FILEID1/FILEID2/FILEID3/FILEID4
TEST
(MYUSERCODE)TEST/=

Semantics

According to system-wide definition, a ⟨*file specifier*⟩ consists of from 1 to 14 ⟨*identifier*⟩s of from 1 to 17 characters each, separated by slashes (/). If there are n separated ⟨*identifier*⟩s in a ⟨*file specifier*⟩, the first n-1 ⟨*identifier*⟩s specify directory names and the last ⟨*identifier*⟩ specifies the file name.

The Binder accepts the equal sign (=) character in place of the file name (last ⟨*identifier*⟩), with the following meaning: when the ⟨*file specifier*⟩ is used in binding a subprogram, the subprogram name will be substituted for the equal (=) character in the ⟨*file specifier*⟩, and the Binder will attempt to bind the given subprogram from the file whose title is formed by the substitution. For example, if the Binder attempts to bind a subprogram named "S" using the ⟨*file specifier*⟩ "A/B/=", the Binder looks for a file titled "A/B/S".

The "bridge" over the integer 12 appearing in the syntax diagram is used to illustrate that a maximum of 13 identifiers may be specified. The "switch-back" path may be crossed a maximum of 12 times.

IDENTIFIER

Examples

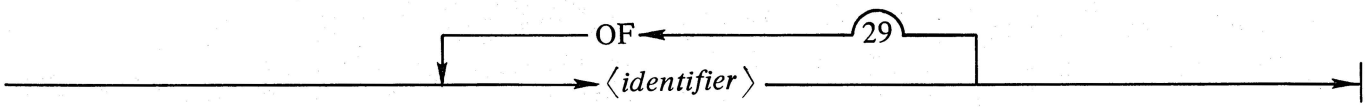
A/B/C
SUBPROG
"*SYSTEM"

Semantics

An ⟨*identifier*⟩ is defined as any sequence of characters A-Z, 0-9, ., -, @, /, $, #, or any character(s) contained within a pair of quotation marks (" ").

SUBPROGRAM IDENTIFIER

Syntax



Examples

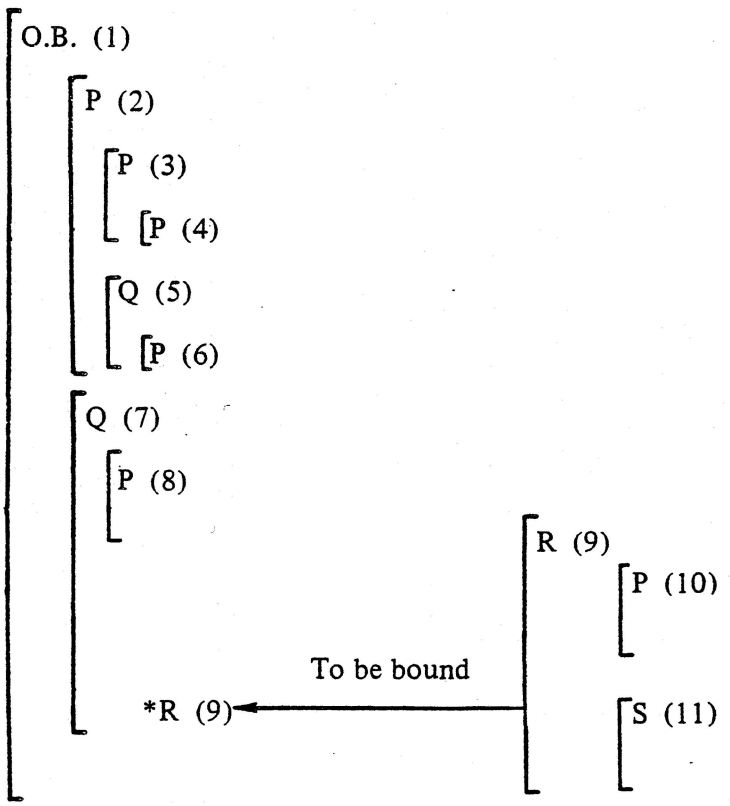
P
SUBR
P OF Q
TEST-5 OF TEST-4 OF TEST-3

Semantics

A ⟨*subprogram identifier*⟩ is the name of a subprogram. In cases where more than one subprogram have the same name, one subprogram can be differentiated from another by the use of qualifiers. A subprogram is qualified by the name(s) of the subprogram(s) within which the subprogram is nested. A ⟨*subprogram identifier*⟩ may contain a maximum of 29 qualifiers to uniquely identify it; however, the qualification may not skip a level of nesting. When a ⟨*subprogram identifier*⟩ is used in a ⟨*binder statement*⟩, the ⟨*binder statement*⟩ is applicable to all subprograms which fit the qualification of the ⟨*subprogram identifier*⟩.

In figure 4-1, the nesting structure of a program containing one external subprogram R, plus the structure of the separately compiled subprogram R is illustrated. As R is bound into the Host, all ⟨*subprogram identifier*⟩s become applicable to subprograms nested within R as well as to those subprograms initially residing in the Host.

For the purpose of illustration in figure 4-1, each subprogram is given a number so that it may be uniquely identified. "O.B." is the name given to the unnamed outer block so that it may be used as a qualifier of a subprogram.



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

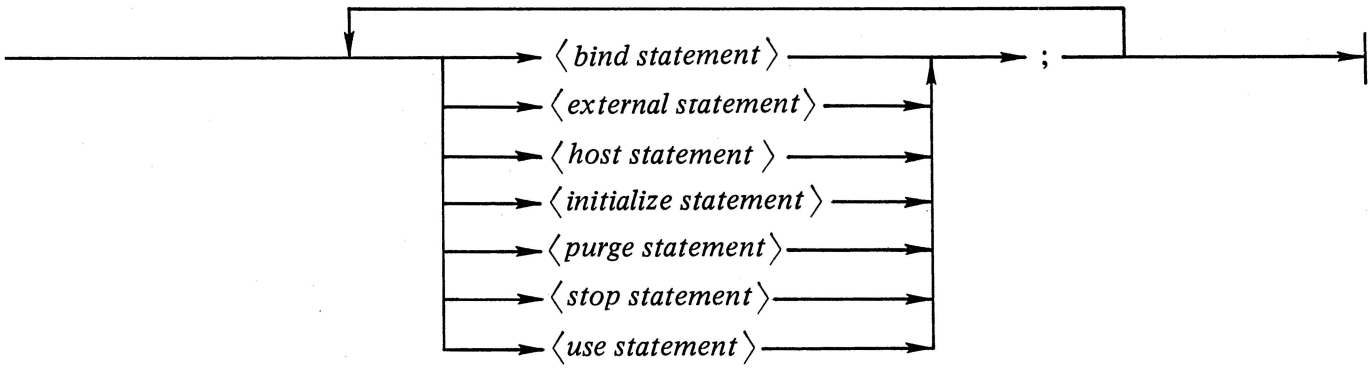
*Declared as EXTERNAL

Figure 4-1. Program Nesting Structure

5. BINDER STATEMENTS

BINDER STATEMENTS

Syntax



Examples

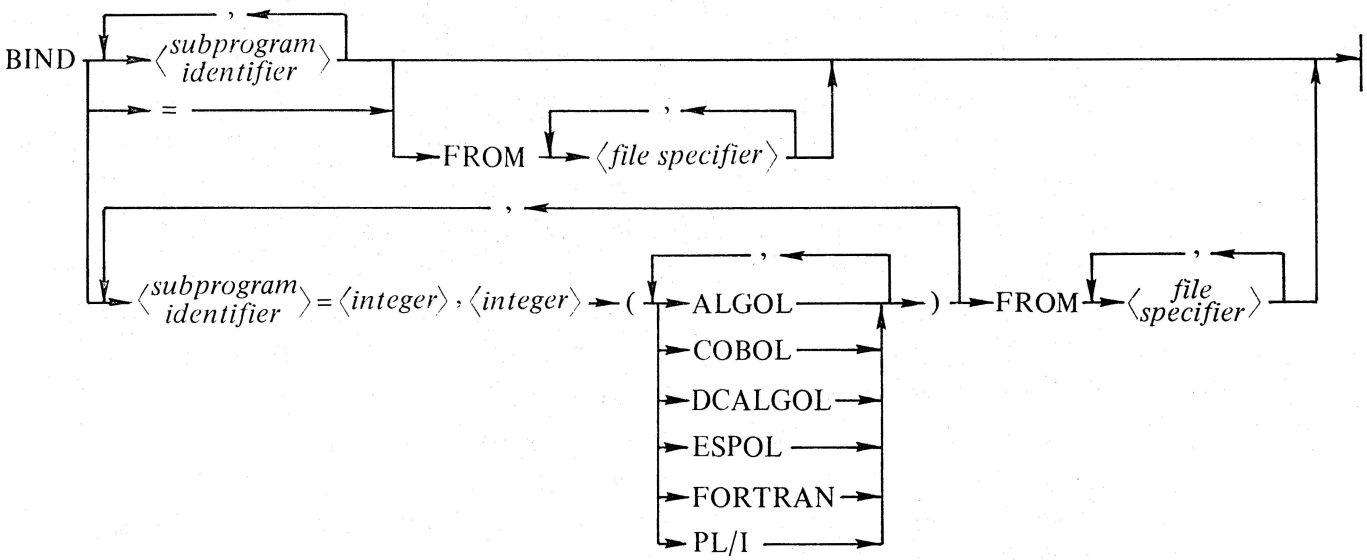
```
BIND SUBA FROM MAIN/PROG;
EXTERNAL SUBB;
HOST IS MY/HOST;
INITIALIZE SUBA = (0,30);
PURGE SUBC/MAIN;
STOP;
USE AFILE FOR BFILE;
```

Semantics

A ⟨*binder statement*⟩ may be written in free form on one or more cards. Columns 73-80 are ignored by the binder. A percent sign (%) appearing in any column from 1 through 72 of a card image indicates that the remaining columns of the card are to be ignored by the Binder.

BIND STATEMENT

Syntax



Semantics

A ⟨*bind statement*⟩ specifies the name(s) of a subprogram(s) to be bound or the title of the file(s) in which the subprogram(s) may be found, or both. If a subprogram is external to the Host file (i.e., not bound), the ⟨*bind statement*⟩ is used primarily to indicate the file in which the subprogram may be found, since an attempt would be made to bind it even if no ⟨*bind statement*⟩ were given. If a subprogram is not external to the Host file (i.e., already bound either by the Binder or compiler), and the subprogram is specified in a ⟨*bind statement*⟩, the Binder replaces the existing subprogram with a new version of the subprogram.

⟨*bind statement*⟩ constructs are provided through which a user can communicate necessary subprogram and subprogram file information to the Binder. The "BIND ⟨*subprogram identifier*⟩ = " construct used for intrinsic binding is illustrated only for completeness of the ⟨*bind statement*⟩ syntax. Refer to section 8, **INTRINSIC BINDING**, for the complete discussion and syntax of this construct.

BIND ⟨*subprogram identifier*⟩ . . .

Examples

```
BIND SUBA;
BIND SUBA FROM A/=;
BIND SUBA, SUBB FROM A/=;
BIND SUBA,SUBB,SUBC FROM A/=, B/=;
BIND SUBA OF SUBX,SUBB OF SUBY FROM LIB/FLE;
BIND SUBA FROM LIB/FLE;
```

The "BIND ⟨*subprogram identifier*⟩" construct directs the Binder to bind specified subprogram(s) into the Host. The ⟨*subprogram identifier*⟩ may specify either resolved or unresolved subprograms. Subprograms declared EXTERNAL (unresolved) in the Host file may be (but need not be) specified in a ⟨*bind statement*⟩; default action is usually sufficient to bind such subprograms.

The "FROM ⟨*file specifier*⟩" feature of the ⟨*bind statement*⟩ allows the user to specify files in which the Binder is to locate subprograms to be bound. If a ⟨*file specifier*⟩ is provided in a ⟨*bind statement*⟩, the Binder examines each file specified, in its order of appearance, to determine if the file contains the subprogram(s) to be bound.

An equal sign (=) appearing as the rightmost ⟨*identifier*⟩ of the ⟨*file specifier*⟩ causes the Binder to replace the equal sign (=) with the name of the subprogram to be bound. For example, a ⟨*bind statement*⟩ of the form "BIND SUBA FROM A/=;" instructs the Binder to locate a file titled "A/SUBA" and from this file bind subprogram SUBA into the specified Host.

If a complete file title is given as a ⟨*file specifier*⟩, only one ⟨*subprogram identifier*⟩ should be specified in the ⟨*bind statement*⟩ or, the ⟨*file specifier*⟩ should name a library file (an ALGOL, FORTRAN, or ESPOL program compiled with the LIBRARY option SET).

If a ⟨*file specifier*⟩ is not included in a ⟨*bind statement*⟩, the Binder will attempt to locate the containing file by using the ⟨*file specifier*⟩ of the "BIND =" ⟨*bind statement*⟩.

BIND = . . .

Examples

```
BIND = FROM THISFILE/=;
BIND = FROM A/=,B/=,C/=;
BIND = FROM A/B/C;
BIND = FROM A/B,B/=, FILEID1/FILEID2/=;
BIND = FROM=;
```

The "BIND =" construct designates files from which the Binder may bind subprograms as they are needed. This construct, of itself, causes no subprograms to be bound. If a subprogram is external or if a ⟨*subprogram identifier*⟩ is specified in a ⟨*bind statement*⟩ and no ⟨*file specifier*⟩ was given in the ⟨*bind statement*⟩, the Binder will attempt to locate the containing file by using the ⟨*file specifier*⟩s declared in this construct.

An equal sign (=) appearing in place of a ⟨*subprogram identifier*⟩ directs the Binder to replace the equal sign (=) with the title of the subprogram to be bound. An equal sign (=) appearing in a ⟨*file specifier*⟩ is also replaced with the subprogram title.

If a "BIND =" statement is required for proper binding, but has not been included in the card input deck, the Binder will create one using the Host file title. An equal sign is substituted for the last file identifier in the Host file title to create a "BIND =" ⟨*bind statement*⟩. For example, given a Host file title of "THIS/IS/MY/HOST," a ⟨*bind statement*⟩ of the form "BIND = FROM THIS/IS/MY/=;" would be created.

Having created a "BIND =" statement, the Binder attempts to locate the subprogram in the file specified by the generated bind statement. Consider the following example: PF is a subprogram to be bound and the Binder has created a ⟨*bind statement*⟩ of the form "BIND = FROM THIS/IS/MY/=;". The Binder, in order to locate the subprogram file containing PF, replaces the equal sign (=) with the subprogram name PF, which in turn causes the ⟨*bind statement*⟩ to assume the form "BIND PF FROM THIS/IS/MY/PF;". The Binder locates the file titled "THIS/IS/MY/PF" and binds from this file the subprogram named PF.

In certain cases of intrinsic binding, where no Host file is used, the title of the code file, as specified in the compile statement, is used in place of the title of the Host file to create a "BIND =" ⟨*bind statement*⟩. The same procedure is followed by the Binder when it creates a "BIND =" ⟨*bind statement*⟩ by means of a code file title; an equal sign (=) replaces the last identifier of the code file title, the subprogram name replaces the equal sign (=), the Binder locates the file specified by the new ⟨*bind statement*⟩ and binds the subprogram into the file specified on the compile control card.

If more than one BIND= statement is given to the BINDER, all but the last one is ignored and an appropriate warning message is given.

If all attempts to locate the containing file fail, the Binder emits an appropriate message and continues to look for other subprograms to bind.

The following examples illustrate how the two constructs previously discussed can be used to specify subprogram files. The ⟨*bind statement*⟩s given below all have the same net effect.

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

b.
```
BIND = FROM FILEID/=;
```
(P, Q, and R are external to the Host and therefore bound by default).

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

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

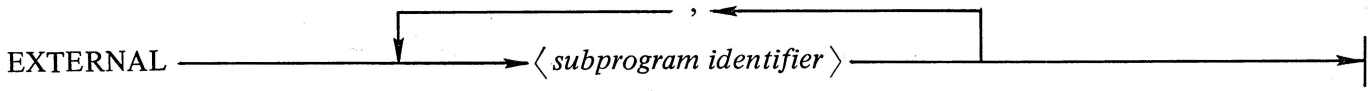
Pragmatics

File-naming conventions make possible the compilation of multiple subprograms into separate code files during a single compilation, as well as the specification of directory names in a ⟨*bind statement*⟩, so that the same subprogram file(s) may be found by the Binder in a later binding process. When compiling one or more subprograms, the FORTRAN compiler (when the SEPARATE option is set); the PL/I compiler (when the MULTIPLE option is set), the ESPOL compiler, and the ALGOL compiler create one or more code files with a title that is formed by replacing the last ⟨*file identifier*⟩ of the code file title on the compile card with the name of the separate subprogram. The title of the Host file is not changed. In COBOL the name of the subprogram is taken from the last ⟨*identifier*⟩ of the compiler-generated code file. Thus, in each instance the name of a separately compiled subprogram is the same as the last ⟨*file identifier*⟩ of the code file in which it resides.

In ALGOL, FORTRAN, and ESPOL, a LIBRARY option is available which causes the compiler to place all subprograms compiled in a single compilation into one code file. The title of the code file is not changed, but remains as specified on the compile control card. The LIBRARY option is automatically set to TRUE when compilation of an ALGOL or FORTRAN program is initiated by CANDE.

EXTERNAL STATEMENT

Syntax



Examples

```
EXTERNAL P;
EXTERNAL P OF Q OF R;
EXTERNAL SUBR1, SUBR2, TEST-Z;
```

Semantics

An ⟨*external statement*⟩ directs the Binder not to bind specified subprograms. If a subprogram is external to the Host, it will be left as an unresolved external reference when its name appears in an ⟨*external statement*⟩.

Pragmatics

Example 1

```
BIND SUBR;
EXTERNAL SUBR;
```

Example 2

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

Potential conflicts exist between the two ⟨*binder statement*⟩s shown in example 1, as well as the three ⟨*binder statement*⟩s shown in example 2. These conflicts are resolved in the following manner. When the Binder examines a subprogram named in the Host for possible binding, it checks to see if any ⟨*bind statement*⟩s or ⟨*external statement*⟩ s apply to the subprogram. Given that more than one statement potentially applies to the given subprogram, all but one of the statements is eliminated according to the following priority scheme.

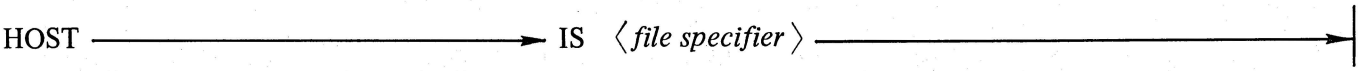
a. The statement containing the ⟨*subprogram identifier*⟩ with the most qualifiers is selected so long as the qualification matches the environment of the given subprogram.

b. In cases where a ⟨*bind statement*⟩ and ⟨*external statement*⟩ contains the same number of qualifiers, the ⟨*external statement*⟩ is selected.

c. In cases where more then one ⟨*bind statement*⟩ applies, each having the same number of qualifiers, the last ⟨*bind statement*⟩ is selected.

HOST STATEMENT

Syntax



Examples

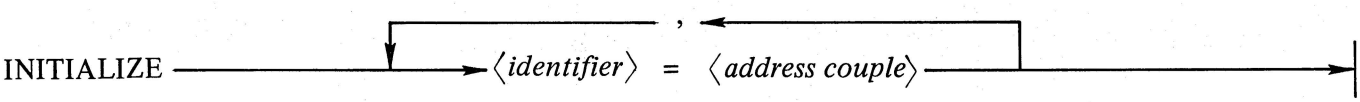
```
HOST IS MY/HOST;
HOST IS *SYSTEM/PL/I;
HOST IS (MYUSERCODE)HOST/FILE;
```

Semantics

The ⟨*host statement*⟩ specifies the title of the Host file. The ⟨*host statement*⟩ overrides any label-equating involving the Binder file HOST. If more than one ⟨*host statement*⟩ appears in the card input file, only the last ⟨*host statement*⟩ will be effective.

INITIALIZE STATEMENT

Syntax



Examples

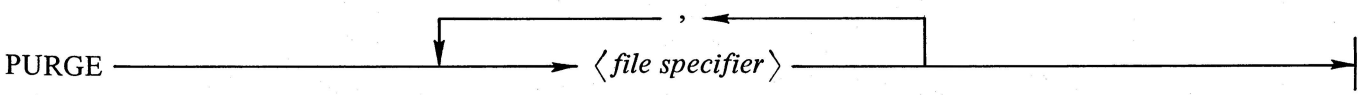
```
INITIALIZE A = (0,50);
INITIALIZE BLOCKEXIT = (O,B);
```

Semantics

The ⟨*initialize statement*⟩ is allowed only in MCP and intrinsic binding. This statement is used in intrinsic binding when an MCP global item at a fixed location is referenced by an intrinsic written in a language other than ESPOL. Since a non-ESPOL compiler does not have the correct address couple of the MCP global item at compile time, the Binder allows the user to specify the correct address couple for the MCP global item. Extreme care should be taken in the use of this statement; otherwise unpredictable results can occur.

PURGE STATEMENT

Syntax



Examples

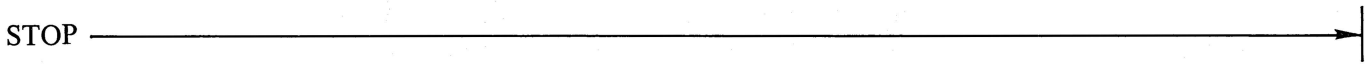
```
PURGE SEP/FILE;
PURGE FILE1/=;
PURGE =;
PURGE FILE1/=, FILE2/=;
```

Semantics

The ⟨*purge statement*⟩ causes the indicated files to be removed from the disk after the Binder has finished its processing. The Binder will not remove files which it does not access during the binding process, even though they may be specified in a ⟨*purge statement*⟩.

STOP STATEMENT

Syntax



Example

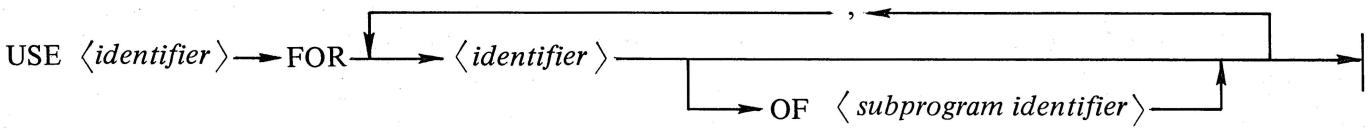
```
STOP;
```

Semantics

The ⟨*stop statement*⟩ indicates the end of the input card deck. All cards following the ⟨*stop statement*⟩ are ignored by the Binder. This provides a convenient means for the user to store additional Binder input cards in one deck.

USE STATEMENT

Syntax



Examples

```
USE A FOR B;
USE ALGOLARRAY FOR /COMMON/;
USE A FOR B,C,D;
USE X FOR Y OF SUBR1;
USE Z FOR Y OF SUBR2;
```

Semantics

The ⟨*use statement*⟩ provides a means of matching ⟨*identifier*⟩s of separately compiled subprograms with ⟨*identifier*⟩s in a Host. The first ⟨*identifier*⟩ (given just after the USE) is the ⟨*identifier*⟩ contained in the Host. The ⟨*subprogram identifier*⟩*s* specified after the FOR are ⟨*identifier*⟩s which are referenced by subprograms to be bound into the Host. The ⟨*identifier*⟩ in the Host does not change as a result of a ⟨*use statement*⟩. When a ⟨*use statement*⟩ is invoked and the Host ⟨*identifier*⟩ does not exist in the Host, the item which was referenced by the subprogram will be added to the Host and given the name of the Host ⟨*identifier*⟩ as specified in the ⟨*use statement*⟩. The ⟨*identifier*⟩ referenced by the subprogram may be qualified by the subprogram name so that the ⟨*use statement*⟩ will only be invoked when binding the specified subprogram.

Special considerations are necessary when the ⟨*use statement*⟩ concerns the subprogram name itself. According to file-naming conventions (see Pragmatics of the ⟨*bind statement*⟩) in this section, the last ⟨*identifier*⟩ in the title of the separately compiled subprogram is the subprogram name. When the Binder, however, looks for the subprogram file through a ⟨*file specifier*⟩ with an equal sign (=) ending, the Binder will also look for a file title ending with the subprogram name as found in the Host file. For example, assuming that Q is a subprogram contained in a file titled "A/Q", the Binder would not find the correct file through use of the input deck shown in Example 1 below. The ⟨*bind statement*⟩ included in Example 2 is necessary for correct binding.

Example 1

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

Example 2

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

If the Binder is directed to bind a subprogram from a specified file, and after reading the specified file, the Binder discovers that the ⟨*identifier*⟩ of the separately compiled subprogram does not match the ⟨*identifier*⟩ of the subprogram in the Host, a warning message is printed and the Binder creates a ⟨*use statement*⟩ which corrects the ⟨*identifier*⟩ mismatch.

6. INTRALANGUAGE BINDING

Intralanguage binding consists of binding a subprogram(s) and a Host where both the subprogram(s) and the Host have been written in the same programming language. This section discusses the various intralanguage binding techniques required to perform intralanguage binding for ALGOL, for COBOL, for FORTRAN, and for PL/I.

ALGOL INTRALANGUAGE BINDING

ALGOL intralanguage binding consists of binding an ALGOL procedure into an ALGOL Host.

ALGOL Host

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

ALGOL Subprogram

An ALGOL subprogram is an ALGOL procedure.

A procedure to be bound into a Host must match the procedure declaration in the Host as to type of procedure and number and type of parameters. Additionally, the procedure must be compiled at the correct execution level: a level which matches its defined execution level within the Host. The LEVEL option is used to set the desired execution level of the procedure to be bound.

If unmatched ⟨*subprogram identifier*⟩s exist in the Host and in the procedure to be bound, a Binder ⟨*use statement*⟩ should be declared to correct this mismatch. Parameter names declared in the procedure need not be the same as those declared in the Host. A separately compiled procedure may reference any item declared in the Host that is global to the execution level of that procedure. This includes items at intermediate lexicographical levels.

Any item declared after the body of the procedure in the Host program may also be referenced by the separately compiled procedure. Therefore, a Host program that contains a separately compiled and bound procedure may produce different results than the same program compiled fully by the ALGOL compiler.

For the most part, ALGOL, DCALGOL, and DMALGOL are considered to be the same language. Mixing the procedures of these three languages through binding is considered to be intralanguage; however, the BINDER views these languages as distinct when dealing with formal procedure parameters. As with all other interlanguage binding, passing formal procedure parameters between these languages is prohibited by the BINDER.

Separate Compilation Methods

All globals, referenced by a separately compiled procedure, must be declared in some form during the compilation of the separate procedure to ensure that no undeclared global ⟨*identifier*⟩s exist within the separate procedure. Two methods are available for declaring global items: the brackets method and the Info File method discussed below.

BRACKETS METHOD FOR DECLARING GLOBALS

Globals referenced by a separate procedure must be declared prior to the procedure itself. These globals must be enclosed in brackets ([ ]) and placed prior to the procedure. The following is an example of a "bracketed" global declaration.

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

If more than one procedure is compiled during a single compilation, all globals referenced by all procedures must be declared within the same set of brackets and must appear before the first procedure to be compiled.

The declaration of a global item, in most cases, is similar to a normal item declaration except for the following:

a. An array need only be declared with LOWER BOUNDS.

b. SWITCH items may be declared without the corresponding SWITCH LISTs.

c. A LABEL item will be interpreted as an illegal new global (see below) unless a BAD GOTO to that label appeared in the host.

INFO FILE METHOD FOR DECLARING GLOBALS

The use of Info Files provides a second method for declaring global items. (A complete description of Info Files, the DUMPINFO option, and the LOADINFO option is contained in the ALGOL Language Reference Manual, form 5011760.)

A DUMPINFO option card is used to place global declarations in the Info File. The DUMPINFO card may appear at any point within the symbolic and causes all globals specified prior to its appearance to be placed in the Info File for future reference. For example, if a DUMPINFO card is placed just prior to the last END card of a program, all global items specified in that program would be placed in the Info File.

When separately compiling procedures, one may recover the "saved" global declarations by using a LOADINFO option card. As with the Bracketed method, the LOADINFO option must be specified before the first procedure to be compiled.

The Info File method of declaring globals, in combination with the Bracketed method, is used for the purpose of adding additional globals to the Info File without recompiling the original program that created the Info File. This method of adding globals is accomplished by placing the LOADINFO option within the brackets prior to the first global declaration.

Info Files created by different versions of the ALGOL compiler are not compatible. For example, an Info File created by the II.5 ALGOL compiler may not be used by the II.6 ALGOL compiler.

ADDING NEW GLOBALS

When binding a procedure into a Host, if a referenced global item is not declared in the Host, the Binder adds the item to the Host as a "new" global. New globals are added at the global level of the Host. The following rules apply when adding new globals to a Host program.

a. The following variable types may not be added as new globals:

FORMAT
LABEL
LIST
PICTURE
"SWITCH TYPE ITEMS"

b. An array added as a new global must be declared with both UPPER BOUNDS and LOWER BOUNDS.

c. A procedure to be added to the Host is added as if it had been declared EXTERNAL.

d. A file to be added to the Host is "stripped" of any file attributes prior to its becoming a new global. All necessary file attributes must be label equated.

SEPCOMP

The SEPCOMP dollar option card is an extremely valuable tool which combines the features of the dollar options GO TO, LOADINFO, DUMPINFO, and AUTOBIND. The SEPCOMP procedure is designed to allow easy recompilation and binding of procedures contained within one large symbolic.

Preparation for the SEPCOMP process is made by setting the dollar option MAKEHOST during the compilation of the Host file. Following the word MAKEHOST, and enclosed within a single set of parentheses is a list of those procedures which may take advantage of the SEPCOMP facility. For example, PASSI and PASSII are two large procedures that contain many small procedures. If the dollar option card $MAKEHOST (PASSI, PASSII) is included during the compilation, it is possible to SEPCOMP a procedure within PASSI or PASSII without recompiling all of PASSI or PASSII.

The MAKEHOST option facilitates SEPCOMP for all global level items within the Host even when they are not specified within a set of parentheses.

Once the Host has been compiled using the MAKEHOST option, various procedures within the Host may be modified by the use of SEPCOMP. Basically, the user arranges the patch deck needed to perform the SEPCOMP as if performing a full compilation of the symbolic. The option SEPCOMP is SET in the beginning of the patch deck. The ALGOL compiler determines which procedures are to be recompiled by means of the specification given in the patch deck. Once the recompiling procedures have been performed, the Binder is invoked, replacing the specified procedures within the Host with the new patched versions. In each case only the highest level procedure for which SEPCOMP is facilitated and which is also included in the patch deck is compiled and bound.

Example Program

The following is an example of ALGOL intralanguage binding. When executed, the output:

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

is produced by means of the following three decks:

a. The ALGOL Host is compiled as follows:

```
〈I〉    COMPILE HOST ALGOL LIBRARY;
〈I〉    DATA
        BEGIN
        FILE LINE(KIND=PRINTER,MAXRECSIZE=22) ;
        ARRAY BUFFER[ 0:2,0:8 ] ;
        REAL I ;
        PROCEDURE PRINTIT ; EXTERNAL ;
```

```
        FOR I:=0 STEP 1 UNTIL 8 DO
            BUFFER[0,I]:=BUFFER[1,I]:=BUFFER[2,I]:=" " ;
        FOR I:=0,1,2 DO
            BUFFER[I,01]:=BUFFER[I,2]:=BUFFER[I,4]:="*" ;
        BUFFER[1,1]:= "*" ;
        PRINTIT ;
        END.
⟨I⟩
```

b. The subprogram is compiled as follows:

```
⟨I⟩  COMPILE PRINTIT ALGOL LIBRARY;
⟨I⟩  DATA
     [ ARRAY BUFFER[0,01] ; REAL I, J ; FILE LINE ;]
     PROCEDURE PRINTIT ;
     BEGIN
     FOR J: = 0 STEP 1 UNTIL 2 DO
         WRITE(LINE,<3A1,X1,3A1>, FOR I;=0 STEP 1 UNTIL 5
         DO BUFFER[J,I]);
     END.
⟨I⟩
```

c. The files are then bound as follows:

```
⟨I⟩  BIND BELL/DONG BINDER;
⟨I⟩  END JOB
```

COBOL INTRALANGUAGE BINDING

COBOL intralanguage binding consists of binding a COBOL subprogram into a COBOL Host.

COBOL Host

A COBOL Host is a COBOL program compiled at its default level (2).

To describe an EXTERNAL procedure to be bound in a COBOL Host, the Host must contain an EXTERNAL procedure declaration in the DECLARATIVES portion of the PROCEDURE DIVISION. Parameters, if any, must be declared in the LOCAL-STORAGE (LD entry) in the DATA DIVISION and must be identified as to whether they are passed by REFERENCE or CONTENTS.

Within the SPECIAL-NAMES paragraph, the CODE FILE TITLE IS MNEMONIC NAME option is used to supply the Binder with the external name of the subprogram to be bound. An explicit ⟨*bind statement*⟩ overrides the SPECIAL-NAMES paragraph. Thus, the Binder attempts to bind the subprogram from a code file whose title is specified according to the following precedence:

a. explicit ⟨*bind statement*⟩.

b. SPECIAL-NAMES paragraph.

Once bound, the subprogram(s) are activated by the ENTER or CALL verb. (The two verbs are synonymous; ENTER can be substituted for CALL in the following discussion or example.) When the CALL verb is used, the name immediately following the CALL must be the name of the section in the DECLARATIVES which contains the procedure description of the external subprogram. The name which the Binder uses in

processing the external subprogram is the section-name in which the subprogram is declared EXTERNAL. All ⟨*binder statement*⟩s pertaining to an external subprogram must reference its corresponding section-name.

COBOL Subprogram

Subprograms to be bound to a Host must be compiled at a level compatible with their usage in the Host. If a subprogram is declared directly in the Host, the LEVEL option must be set to 3 during the compilation of that subprogram. All subprograms to be bound must be compiled at one level higher than the level of the subprogram in which they are declared. For example, a subprogram to be bound to a level 3 subprogram must be compiled at level 4.

If any parameters are passed to a COBOL subprogram, they must be described in the subprograms. Such parameters must also be specified following the keyword USING in the PROCEDURE DIVISION heading.

Global Declarations

Any variable that is declared in the WORKING-STORAGE SECTION (and can be passed or received as a parameter) can be declared global by using the GLOBAL clause. In addition to the above-mentioned variables, untyped procedures, files, and direct files are the only valid global items that may be declared. Several examples of declaring variables (WORKING-STORAGE SECTION) are provided below.

a. **77 GLASTATUS GLOBAL COMP-1 PIC 9(11).**

b. **77 BL-EVNT GLOBAL EVENT.**

c. **77 GL-SWFL INDEX FILE GLOBAL.**

d. **01 GL-RCD RECORD AREA GLOBAL OCCURS 10 SZ 180.**

e.
```
01 GL-EBCRAY GLOBAL.
    03 CMP-ITE COMP PIC 9(11)
        OCCURS 100 INDEXED BY 11.
```

If most or all variables declared in the WORKING-STORAGE SECTION are global, the dollar option GLOBAL is used. This option may be set throughout the compilation and it only affects those variables which are candidates for global declarations. The GLOBAL option only affects those items declared in the WORKING-STORAGE SECTION, it does not affect items declared in any other section. The LOCAL or OWN option may be used to override the GLOBAL option as shown in the following example.

```
$ SET GLOBAL
77 G1 COMP-1 PIC 9(11).
77 G2 COMP-1 PIC 9(11).
77 L1 LOCAL COMP-1 PIC 9(11).
01 G3.
    03 FLD OCCURS 10 INDEXED BY I.
```

In the above example, items G1, G2, and G3 are declared GLOBAL; L1 and I are declared LOCAL.

Own Declarations

COBOL programs compiled at level 3 or higher may declare certain variables to be OWN. OWN variables retain their initial value or state throughout repeated exit and re-entry of the subprogram in which they are declared. Any item capable of being declared in the WORKING-STORAGE SECTION, except for direct

switch files, can be declared OWN either by using the OWN clause or the OWN dollar option. All related index names for OWN items are also OWN; redefinitions of OWN items are implicitly OWN and need not be specified in the OWN clause. Use of the dollar option OWN throughout the compilation causes all variables declared in the WORKING-STORAGE SECTION, except for direct files, to be declared OWN. The OWN option may be overridden by the LOCAL or GLOBAL clause for any individually specified item.

Example Program

The following is an example of COBOL intralanguage binding. When executed, the bound file produces the output:

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

by means of the following three decks.

a. The COBOL Host is compiled to library as follows:

```
〈I〉   COMPILE COBOL/HOST COBOL LIBRARY
〈I〉   DATA
           IDENTIFICATION DIVISION.
           PROGRAM-ID. HOST
           ENVIRONMENT DIVISION.
           CONFIGURATION SECTION.
           SOURCE-COMPUTER. B-6700.
           OBJECT-COMPUTER. B-6700.
           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 SIZE 36.
           WORKING-STORAGE SECTION.
           01 ORIG SIZE 36.
           01 NEW SIZE 36.
           LOCAL-STORAGE SECTION.
           LD PARMS.
           01 A SIZE 36 REF.
           01 B SIZE 36 REF.
           PROCEDURE DIVISION.
           DECLARATIVES.
               S1 SECTION.
               USE EXTERNAL TO-BE-CALLED AS PROCEDURE WITH
           PARMS USING A B.
           END DECLARATIVES.
           THE 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.
⟨I⟩
```

b. The COBOL subprogram is compiled to library as follows:

```
⟨I⟩   COMPILE COBOL/PROG COBOL LIBRARY ;
⟨I⟩   DATA
$ SET LEVEL 3
      IDENTIFICATION DIVISION.
      PROGRAM-ID.  ARRAY/MIXER.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SOURCE-COMPUTER.  B-6700.
      OBJECT-COMPUTER.  B-6700.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01 X REF.
          03 ONE SIZE 5.
          03 SECOND SIZE 5.
          03 THIRD SIZE 5.
          03 FOURTH SIZE 5.
          03 FIFTH SIZE 5.
          03 SIXTH SIZE 5.
          03 SEVENTH SIZE 5.
          03 EIGHTH SIZE 1.
      01 Y REF.
          03 FIRS SIZE 5.
          03 SECON SIZE 5.
          03 THIR SIZE 5.
          03 FOURT SIZE 5.
          03 FIFT SIZE 5.
          03 SIXT SIZE 5.
          03 SEVENT SIZE 5.
          03 EIGHT SIZE 1.
      PROCEDURE DIVISION USING X Y.
      THAT 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.
⟨I⟩
```

c. The Host and subprogram are then bound and executed as follows:

```
⟨I⟩    BIND COBOL/EXAMPLE BINDER
⟨I⟩    DATA
HOST IS COBOL/HOST ;
USE S1 FOR PROG ;
BIND S1 FROM COBOL/PROG ;
⟨I⟩    END JOB
```

FORTRAN INTRALANGUAGE BINDING

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

FORTRAN Host

A FORTRAN Host is a FORTRAN main program. The Host may contain one or more subroutines or functions, provided they are compiled with the Host.

FORTRAN Subprogram

A FORTRAN subprogram is any FORTRAN subroutine or function. Any subprogram bound into a Host must match its invocation(s) in the Host as to number and type of parameters.

If the ⟨*subprogram identifier*⟩ does not match its invocation as specified in the Host, a Binder ⟨*use statement*⟩ must be specified to correct the mismatch. In addition, the subprogram must be compiled at an execution level consistent with its execution level within the Host.

In cases where an entry point is referenced by a Host, but the corresponding subprogram is not referenced, the file in which the entry point is located must be explicitly specified to the Binder in a ⟨*bind statement*⟩.

Example Program

The following is an example of FORTRAN intralanguage binding. When executed, the output:

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

is produced by means of the following three decks:

a. The Host is compiled as follows:

```
⟨I⟩    COMPILE FORTRAN/HOST FORTRAN LIBRARY;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
⟨I⟩
```

b. The subprogram is compiled as follows:

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

c. The two files are bound as follows:

```
⟨I⟩     BIND SINE BINDER;
        HOST IS FORTRAN/HOST;
⟨I⟩     END JOB
```

PL/I INTRALANGUAGE BINDING

A PL/I subprogram may be bound only to a PL/I Host. The intralanguage binding process consists generally of binding one or more external procedures to a Host. Communication between the procedure is performed through common EXTERNAL declarations, within the procedures, and parameters.

PL/I Host

Any external PL/I procedure is capable of being declared as the Host, provided that the only parameters declared within the external procedure are CHARACTER VARYING or DECIMAL FIXED.

PL/I Subprogram

Any external procedure not specified as the Host is considered to be a subprogram; no parameter type restrictions exist within subprograms.

Static External

If a variable is declared STATIC EXTERNAL in both the Host and external procedure, the initial value established for the STATIC EXTERNAL variable in the Host is the value the variable retains upon binding. A variable declared STATIC EXTERNAL in only the external procedure retains the value established in the external procedure.

A STATIC EXTERNAL variable declared in more than one external procedure and not declared in the Host causes a bind error upon binding.

An example of declaring variables as STATIC EXTERNAL in both a Host and an external procedure is shown below:

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

When the above subprogram is bound and executed, the results obtained are as follows:

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

Any STATIC EXTERNAL CONTROLLED or BASED variable must be, INITIALIZED before it is used in a declaration. For example,

```
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);
```

must be declared as:

```
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 to be produced at compilation time.

Example Program

The following is an example of PL/I intralanguage. When bound and executed this sample produces the following output character string on the SYSPRINT print file: ABCDEFGH.

a. The Host program is compiled.

```
⟨I⟩     COMPILE HOST/HOST  PL/I  LIBRARY; DATA
HOST:PROC;
        DCL  SEPARATE ENTRY  (CHAR(*))EXTERNAL;
        DCL CHR CHAR(8)  INIT('ABCDEFGH');
        CALL SEPARATE(CHR);
END HOST;
⟨I⟩
```

b. The subprogram is compiled.

```
⟨I⟩     COMPILE SEPARATE/SEPARATE PL/I LIBRARY; DATA
SEPARATE:PROC(C);
        DCL  C  CHAR(*);
        PUT LIST(C);
END SEPARATE;
⟨I⟩
```

c. The two files are bound.

```
⟨I⟩     BIND BOUND BINDER LIBRARY
⟨I⟩     DATA
HOST IS HOST/HOST;
BIND SEPARATE FROM SEPARATE/SEPARATE;
STOP;
⟨I⟩     END JOB
```

7. INTERLANGUAGE BINDING

The Binder is capable of binding together subprograms and a Host program written in various languages into one executable program. Of the languages for which binding is defined, table 7-1 illustrates the various combinations of interlanguage binding available. A brief description is provided for each of these combinations. Statements made in this manual concerning ALGOL also apply to the various extensions of ALGOL, such as DCALGOL and DMALGOL. Table 7-2 illustrates the correspondence between variable types in ALGOL, FORTRAN, and COBOL.

Table 7-1. Interlanguage Binding Combinations

Host Program	ALGOL	COBOL	ESPOL	FORTRAN	PL/I
ALGOL		X		X	
COBOL	X			X	
MCP	X		X		
INTRINSIC	X	X	X		
FORTRAN	X	X			
PL/I					

ALGOL-NEWP Interlanguage Binding

The BINDER is capable of binding DCALGOL code into MCP code files compiled in NEWP. Replacement binding is not allowed for procedures in the NEWP host, except for externals that were bound in and need to be rebound.

Because of the interaction with the NEWP and SEPCOMP facility, the old code of the procedures being rebound is retained in the code file (although it is not pointed at and cannot be executed); thus, a substantial number of rebindings can cause the MCP code file to grow undesirably large.

The following restrictions apply to a subprogram being bound in:

a. The subprogram cannot have been compiled in NEWP.

b. The subprogram must have been declared external in the host.

c. The subprogram cannot add any globals to the host or contain any OWN variable declarations.

d. The only global variables that can be referenced by the subprogram are ones that were declared in the outerblock of the MCP.

The BINDER will not handle binding to non-MCP hosts compiled in NEWP.

ALGOL-COBOL Interlanguage Binding

ALGOL-COBOL interlanguage binding consists of binding either an ALGOL subprogram into a COBOL Host or a COBOL subprogram into an ALGOL Host. Restrictions are placed on the passing of global items and parameters between ALGOL and COBOL. These restrictions are discussed in the following paragraphs.

GLOBALS

Global items may be shared between COBOL and ALGOL programs. If a COBOL subprogram is to reference a variable in an ALGOL Host, the item must be declared with the GLOBAL clause or GLOBAL dollar option in the COBOL subprogram. Similarly, an ALGOL variable must be declared global by one of the various methods (such as brackets) when an ALGOL procedure which references a COBOL global is compiled separately.

Table 7-2. Corresponding Variable Types

ALGOL	COBOL	FORTRAN
REAL ARRAY	COMP	REAL ARRAY/COMMON BLOCK
INTEGER ARRAY	COMP	INTEGER ARRAY
BOOLEAN ARRAY	COMP	LOGICAL ARRAY
DOUBLE ARRAY		DOUBLE PRECISION ARRAY
HEX ARRAY	COMP-2	
BCL ARRAY	DISPLAY-1	
ASCII ARRAY	ASCII	
EBCDIC ARRAY	DISPLAY	
REAL	LEVEL 77	REAL
INTEGER	LEVEL 77	INTEGER
BOOLEAN	LEVEL 77	LOGICAL
DOUBLE		DOUBLE PRECISION
FILE	FILE	FILE
PROCEDURE	SECTION	SUBROUTINE
TYPED PROCEDURE		FUNCTION
DIRECT FILE	DIRECT FILE	

PARAMETERS

Certain restrictions are imposed on parameters passed between ALGOL and COBOL: Arrays passed to or received from COBOL subprograms should be declared in the ALGOL program with a lower bound of zero. When a word-oriented (INTEGER, REAL, BOOLEAN) array is passed between ALGOL and COBOL, the array must be declared COMPUTATIONAL in the COBOL program. String arrays may also be passed between ALGOL and COBOL. Direct files and files may be passed between ALGOL and COBOL. If the file is declared to have a label record in COBOL, the ALGOL program must declare two parameters in order to interface with the COBOL program: the file or direct file followed by a pointer by VALUE for the label record array. Procedures may not be passed as parameters between ALGOL and COBOL.

ALGOL-FORTRAN Interlanguage Binding

The Binder is capable of binding either an ALGOL subprogram into a FORTRAN Host or a FORTRAN subprogram into an ALGOL Host. However, there are certain restrictions concerning the global values that are shared by the Host and the separately compiled subprogram. There are also limitations placed on the parameters passed between ALGOL and FORTRAN.

GLOBALS

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

Files

An ALGOL file with a declared internal name of "FILEnn", where nn is a one- or two-digit number (without leading zero) from 1 to 99, is identified as the same file as a FORTRAN file with that number. Thus, output to FILE6 (global) in an ALGOL program and a WRITE(6,1) statement in a FORTRAN subroutine bound to that ALGOL Host will both be on the same file. This also applies to using ALGOL files as variable files within a FORTRAN program. For example, if an ALGOL Host declares a global file FILE12, and a FORTRAN subprogram with the statements:

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

is bound into the ALGOL Host, the ALGOL global file is read.

For printer files it should be noted that the ALGOL print routine performs its carriage control after the actual write has been performed. However, the FORTRAN print routine performs carriage control in the opposite manner. To prevent potential printer problems, dollar option WRITEAFTER should be SET during the compilation of the ALGOL program.

Common Blocks

A FORTRAN common block may be accessed by ALGOL as either a single-precision array, a double-precision array, or both. An ALGOL single- or double-precision array may be accessed by FORTRAN through a common block.

All arrays must be global, and common block names must include slashes, e.g., /.BLNK./ or /ABC/.

To bind an ALGOL subprogram to a FORTRAN Host where a common block is involved, the following procedure may be used:

a. To access a common block as single-precision, declare an ALGOL array and equate it to the FORTRAN common block with a Binder ⟨*use statement*⟩. For example, with a single-precision ALGOL array A, and a FORTRAN common block BLK, the ⟨*use statement*⟩ is:

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

b. To access a common block as double-precision, use the same statement as shown above, except declare the ALGOL array as double-precision.

c. To access a common block as both single-precision and double-precision, declare both a single-precision ALGOL array and a double-precision ALGOL array and equate both arrays to the FORTRAN common block with a Binder ⟨*use statement*⟩. For example, with a single-precision ALGOL array A, a double-precision ALGOL array D, and a FORTRAN common block BLK, the ⟨*use statement*⟩ is: USE /BLK/ FOR A,D;

To bind FORTRAN to an ALGOL Host, in which it is desired to have a FORTRAN common block access ALGOL global arrays, the following techniques may be used.

a. To access a single-precision array through a common block, declare an ALGOL array and equate it 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 ⟨*use statement*⟩ would be:
USE A FOR /BLK/;

b. To access a double precision array through a common block, use the same statement as shown in step "a," except declare the ALGOL array as double-precision.

c. To access an ALGOL array as both single-precision and double-precision through a common block, declare a single-precision ALGOL array, an adjacent double-precision ALGOL copy array, and equate the single-precision array to the common block with a Binder ⟨*use statement*⟩. For example, with an ALGOL single array A, and ALGOL double-precision array D, and a FORTRAN common block BLK, the ALGOL arrays must be declared in the ALGOL program as shown below:

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

and the ⟨*use statement*⟩ is: USE A FOR /BLK/;

FORTRAN references to single-precision items in /BLK/ will be changed to refer to A, and FORTRAN references to double-precision or complex items in /BLK/ will be changed to refer to D. It is not sufficient for D to be merely a copy of A, D must also be declared adjacent to A.

Simulating Common in ALGOL

A FORTRAN common block is a one-dimensional, single-precision array with an adjacent double-precision copy descriptor. The contents of the array can be determined by mapping the elements of the common statement into a contiguous array. This procedure can be simulated in ALGOL as illustrated by 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 have been adjusted so that DA[1] and RA[1] in ALGOL are the same as DA(1) and RA(1) in FORTRAN.

If an ALGOL array is bound to a FORTRAN common block containing a shorter length, the length of the common block will be increased. However, if a FORTRAN common block is matched to a shorter ALGOL array, the array length will not be changed.

Also, a FORTRAN common block may not contain an initial value when bound into an ALGOL array. If the ALGOL array is declared locally in an ALGOL procedure, any values contained in the common block will be lost when the ALGOL procedure exists. An ALGOL array may be bound to a common block containing initial values.

PARAMETERS

When binding between ALGOL and FORTRAN, only simple variables, arrays and labels may be passed as parameters between program units. When any of these are used as parameters, the following special conditions apply:

1. All FORTRAN arrays are one dimensional (in the hardware sense), hence only one dimensional ALGOL arrays (or array rows) may be passed between ALGOL and FORTRAN. When ALGOL arrays are passed to FORTRAN routines, an ALGOL subscript value of zero corresponds to a FORTRAN subscript value of one. When FORTRAN arrays are passed to ALGOL, a FORTRAN subscript value used to evaluate the parameter (the value one is used if a subscript is not specified) corresponds to an ALGOL subscript value of zero.

2. In passing simple variables between FORTRAN and ALGOL, name and value parameters may be mixed. (FORTRAN value is not the same as ALGOL value.) In this case, the following effects will occur:

   a. If FORTRAN calls ALGOL and passes a variable as a parameter, the variable acts like a name parameter for all cases.

   b. If FORTRAN calls ALGOL and passes an expression as a parameter, the expression will act as an ALGOL value parameter for all cases.

   c. If ALGOL calls FORTRAN and passes a name parameter for a value parameter, the parameter will act like a FORTRAN value parameter.

   d. If ALGOL calls FORTRAN and passes a value parameter, the parameter acts like an ALGOL value parameter for all cases.

3. If a FORTRAN subscripted variable is passed to an ALGOL array, the ALGOL array should be declared with an asterisk for the lower bound.

Procedures, functions, and subroutines may not be passed as parameters between ALGOL and FORTRAN.

EXAMPLE OF ALGOL-FORTRAN BINDING

The following is an example of binding a FORTRAN subroutine and an ALGOL procedure into a FORTRAN Host. The output produced by the example is:

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

This output is produced by the following three decks.

a. The FORTRAN Host is compiled first, as shown below.

```
⟨I⟩    COMPILE FORT/HOST FORTRAN LIBRARY
⟨I⟩    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)
       STOP
       END
⟨I⟩
```

b. The ALGOL and FORTRAN subprograms are then compiled by means of the following decks:

1. ALGOL subprogram.

```
⟨I⟩    COMPILE FTEST/ALGOL  ALGOL  LIBRARY
⟨I⟩    DATA
$      SET WRITEAFTER
       [FILE FILE6]
       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;
            WRITE (FILE6,⟨A6, X1, A6⟩, A[0], B[0]);
END
⟨I⟩
```

2. FORTRAN subprogram.

```
⟨I⟩    COMPILE FTEST/FORTRAN FORTRAN LIBRARY
⟨I⟩    DATA
$      SET SEPARATE
       SUBROUTINE SUB(A,B)
       DIMENSION A(1), B(1)
       C = A(1)
       A(1) = B(1)
       B(1) = C
       RETURN
       END
⟨I⟩
```

c. The files are bound and executed by means of the following deck:

```
⟨I⟩    BIND ROUND/PROGM BINDER
⟨I⟩    DATA
BIND = FROM FTEST/=;
⟨I⟩    END JOB
```

COBOL-FORTRAN Interlanguage Binding

COBOL-FORTRAN interlanguage binding consists of binding a COBOL program into a FORTRAN Host, or binding a FORTRAN subprogram into a COBOL Host. The restrictions placed on the passing of parameters and global items shared between these two languages are discussed in the following paragraphs.

GLOBALS

Only files and subprograms (FORTRAN) may be shared globally between COBOL and FORTRAN. Files in FORTRAN are given the internal name FILEnn, where "nn" is a one- or two-digit number (without leading zero) which refers to the unit number in a FORTRAN I/O statement. For example, a FORTRAN WRITE(6,1) writes to COBOL FILE6. A COBOL file-name should be declared accordingly in order to share a common file with FORTRAN.

PARAMETERS

Only simple variables (COBOL 77 COMP-1 level items) and arrays may be passed between FORTRAN and COBOL. COBOL arrays must be declared COMPUTATIONAL, since FORTRAN works only with word-oriented arrays. If an array is passed from FORTRAN to COBOL, the level 01 description for that array must contain a LOWER-BOUNDS clause. A LOWER-BOUNDS clause must also appear in the LOCAL-STORAGE SECTION description of the formal parameters if an array is passed from COBOL to FORTRAN. COBOL always assumes that the lower bounds of an array which is passed or received is zero. Undefined results can occur if a FORTRAN subscripted variable is passed to COBOL when the subscript value is other than zero. Similarly, an array appearing in a FORTRAN common block should not be passed to COBOL unless it is the first array appearing in the common block. Subroutines and functions may not be passed as parameters between COBOL and FORTRAN.

Example of Interlanguage Binding

The following is an example of interlanguage binding. The Host is a FORTRAN program which passes an array as a parameter to a COBOL program which calls an ALGOL procedure, which in turn calls another COBOL program. When executed, the bound file produces the following output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ      0123456789
9876543210      ZYXWVUTSRQPONMLKJIHGFEDCBA
```

a. The FORTRAN Host is compiled and placed in the file FORTRAN/HOST by means of the following deck:

```
?I?   COMPILE FORTRAN/HOST FORTRAN LIBRARY
?I?   DATA
      DIMENSION A(7)
      A(1) = "ABCDEF"
      A(2) = "GHIJKL"
      A(3) = "MNOPQR"
      A(4) = "STUVWX"
      A(5) = "YZ      "
C     NOW CALL THE COBOL PROGRAM
      CALL COBPRO(A)
      STOP
      END
?I?
```

b. The COBOL program called from the FORTRAN Host is then compiled and placed in the file SEP/COBPRO by means of the following deck:

```
?I?   COMPILE SEP/COBPRO COBOL LIBRARY
?I?   DATA
$ SET LEVEL 3
      IDENTIFICATION DIVISION.
      PROGRAM-ID.  NUMBERS.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SOURCE-COMPUTER. B-6700.
      OBJECT-COMPUTER. B-6700.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01  COBARY COMP LOWER-BOUNDS REF.
          02  FILLER SIZE 42.
      01  FAKEOUT REDEFINES COBARY SIZE 42.
          03  FILLER SIZE 30.
          03 NUMB SIZE 12.
      LOCAL-STORAGE SECTION.
      LD PASS.
      01 LARY COMP SIZE 20 REF.
      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.
⟨I⟩
```

c. The ALGOL procedure called from the COBOL program is then compiled and placed in the file SEP/ALG by means of the following deck:

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

d. The COBOL program called from the ALGOL procedure is then compiled and placed in the file SEP/COBPRINT by means of the following deck:

```
⟨I⟩   COMPILE SEP/COBPRINT COBOL LIBRARY
⟨I⟩   DATA
$ SET LEVEL 3
        IDENTIFICATION DIVISION.
        PROGRAM-ID. PRINT/ARRAYS.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. B-6700.
        OBJECT-COMPUTER. B-6700.
        INPUT-OUTPUT SECTION.
        FILE-CONTROL.
            SELECT PR ASSIGN TO PRINTER.
        DATA DIVISION.
        FILE SECTION.
        FD PR.
```

```
01   A COMP REF.
     03 A1 SIZE 21.
     03 A2 SIZE 21.
01   F1 REDEFINES A SIZE 42.
01   B COMP REF.
     03  B1 SIZE 21.
     03  B2 SIZE 21.
01   F2 REDEFINES B SIZE 42.
PROCEDURE DIVISION USING A  B.
CB SECTION.
OPEN-PR.
     OPEN OUTPUT PR.
     WRITE PR-RCD FROM F1.
     WRITE PR-RCD FROM F2.
⟨I⟩
```

e. The four files are the bound and executed with the following Binder deck.

```
⟨I⟩     BIND EXAMPLE/PROG BINDER
⟨I⟩     DATA
HOST IS FORTRAN/HOST;
USE A1 FOR ALG;
BIND A1 FROM SEP/ALG;
BIND = FROM SEP/=,
STOP;
⟨I⟩     END JOB
```

8. INTRINSIC BINDING

An intrinsic file consists of user-written subprograms, commonly referred to as installation intrinsics, and the standard system intrinsics such as SIN, SQRT, and the formatting routine. Intrinsics may only be written in DCALGOL, ESPOL, or COBOL; however, any language for which binding is defined may access an intrinsic file. The following paragraphs provide the information necessary for compiling, binding, and accessing an intrinsic file. The appropriate language manual should be referenced for additional information concerning intrinsics.

NOTE

Because of the conversion to support libraries, BINDER issues the following warning message when requested to bind installation intrinsics:

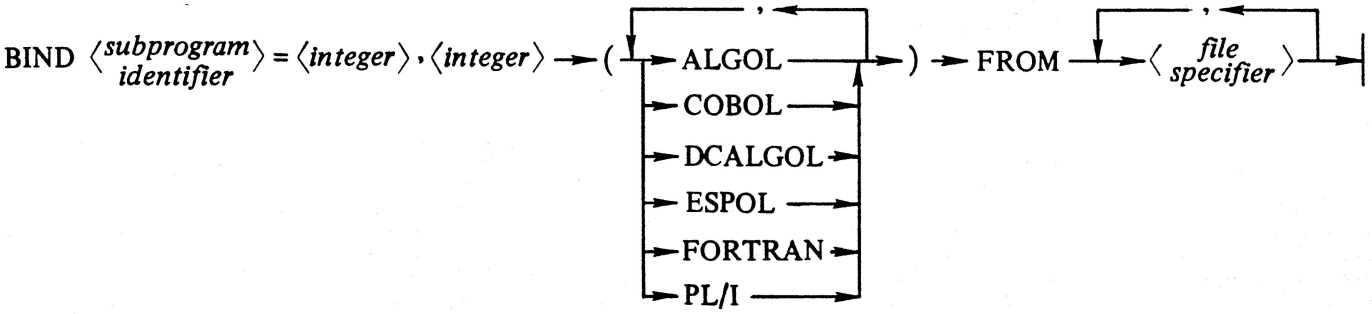
BINDING INSTALLATION INTRINSICS WILL BE DE-IMPLEMENTED ON 34

COMPILING INTRINSICS

An intrinsic may be compiled by the DCALGOL, ESPOL, or COBOL compiler; however, the INTRINSICS dollar option must be set during the compilation. If the intrinsic is compiled by the COBOL compiler, no global items may be referenced. Intrinsics compiled by the DCALGOL or ESPOL compiler may reference only other intrinsics or MCP items.

BINDING INTRINSICS

The ⟨*bind statement*⟩ syntax shown below is used to bind installation intrinsics. Standard system intrinsics, when referenced EXTERNAL in a program, are bound automatically by the Binder and therefore are not specified in a ⟨*bind statement*⟩. The following is the syntax for this statement:



The syntax for the ⟨*bind statement*⟩ is expanded for intrinsic binding. The intrinsic ⟨*bind statement*⟩ provides for the specification of an Intrinsic Number Pair (⟨*integer*⟩ , ⟨*integer*⟩) and a language identifier (those languages ALGOL, COBOL, etc., shown in the syntax) for each intrinsic.

As indicated, an Intrinsic Number Pair consists of two ⟨*integer*⟩s separated by a comma: the first ⟨*integer*⟩ specifies an installation number, the second an intrinsic number. An installation number may range in value between 0 and 2047; however, numbers 0 through 99 are reserved for system use. An intrinsic number may range in value between 0 and 8191. No two intrinsics within an intrinsic file may be given the same Intrinsic Number Pair.

The language identifier referred to allows the user to specify a list of those compilers authorized to reference a given intrinsic.

NOTE

The referencing language is not necessarily the same as that in which the intrinsic is written. The DCALGOL language identifier allows a specified intrinsic to be accessed by the DMALGOL compiler and also by the DCALGOL compiler.

To initially create an intrinsic file, a Binder deck similar in form to that shown below is used. The deck must include:

a. A $ SET INTRINSICS option card.

b. A ⟨*bind statement*⟩ specifying the source file(s) for standard system intrinsics. The "BIND =" statement causes the Binder to look for all the standard intrinsics whose names and intrinsic numbers are tabulated within the Binder.

c. One or more ⟨*bind statement*⟩ specifying the installation intrinsics.

The format of the Binder deck used to create an intrinsic file is:

```
⟨I⟩    BIND SYSTEM/INTRINSICS BINDER LIBRARY
⟨I⟩    DATA
$ SET INTRINSICS
       BIND = FROM INTR/=;
       BIND MYSIN = 101, 1 (ALGOL,FORTRAN) FROM INTL/=;
       BIND COFFEE = 102, 2 (COBOL) FROM POT;
       STOP;
⟨I⟩    END JOB
```

Once an intrinsic is bound into an intrinsic file, the Binder does not allow a user to alter the intrinsic number, type of subprogram, or parameter type by performing replacement binding. Should it become necessary to modify any of these items in an installation intrinsic, the intrinsic must be rebound, with the necessary changes specified, into a "new" intrinsic file. To modify a standard system intrinsic, the Binder internal tables must be updated and a new intrinsic file created.

ACCESSING INTRINSICS

Standard system intrinsics such as SIN and COS are recognized automatically by all compilers as intrinsics. These intrinsics therefore require no unique identification in order for programs to access them.

ALGOL or FORTRAN programs accessing installation intrinsics must be compiled with the INSTALLATION option SET. COBOL and PL/I programs are capable of accessing installation intrinsics automatically.

Appendix A. RESERVED WORDS

The following is a complete list of ⟨*reserved word*⟩s used in the Binder language. These words have special meaning to the Binder and cannot be used in any manner other than their defined meaning.

ALGOL	HOST
BIND	INITIALIZE
COBOL	IS
DCALGOL	NEWP
ESPOL	OF
EXTERNAL	PL/I
FOR	PURGE
FORTRAN	STOP
FROM	USE

Appendix B. BINDER OPTIONS

BINDER CONTROL STATEMENTS

The user is provided with the bind-time ability to control the manner in which the Binder processes the card input deck, the subprogram file(s), and the Host file(s) that it accepts. This feature allows the user to specify the manner in which the Binder is to receive input from these various sources, the consequences of certain syntax errors, and the form of the generated Binder output file(s). The Binder control statement is the medium through which these constraints are communicated to the Binder. Such statements are entered into the Binder by cards in the same manner as the ⟨*binder statement*⟩s. Binder control statements, entered as input to the Binder via option control cards, can occur at any point in the card input deck and must contain only Binder control information.

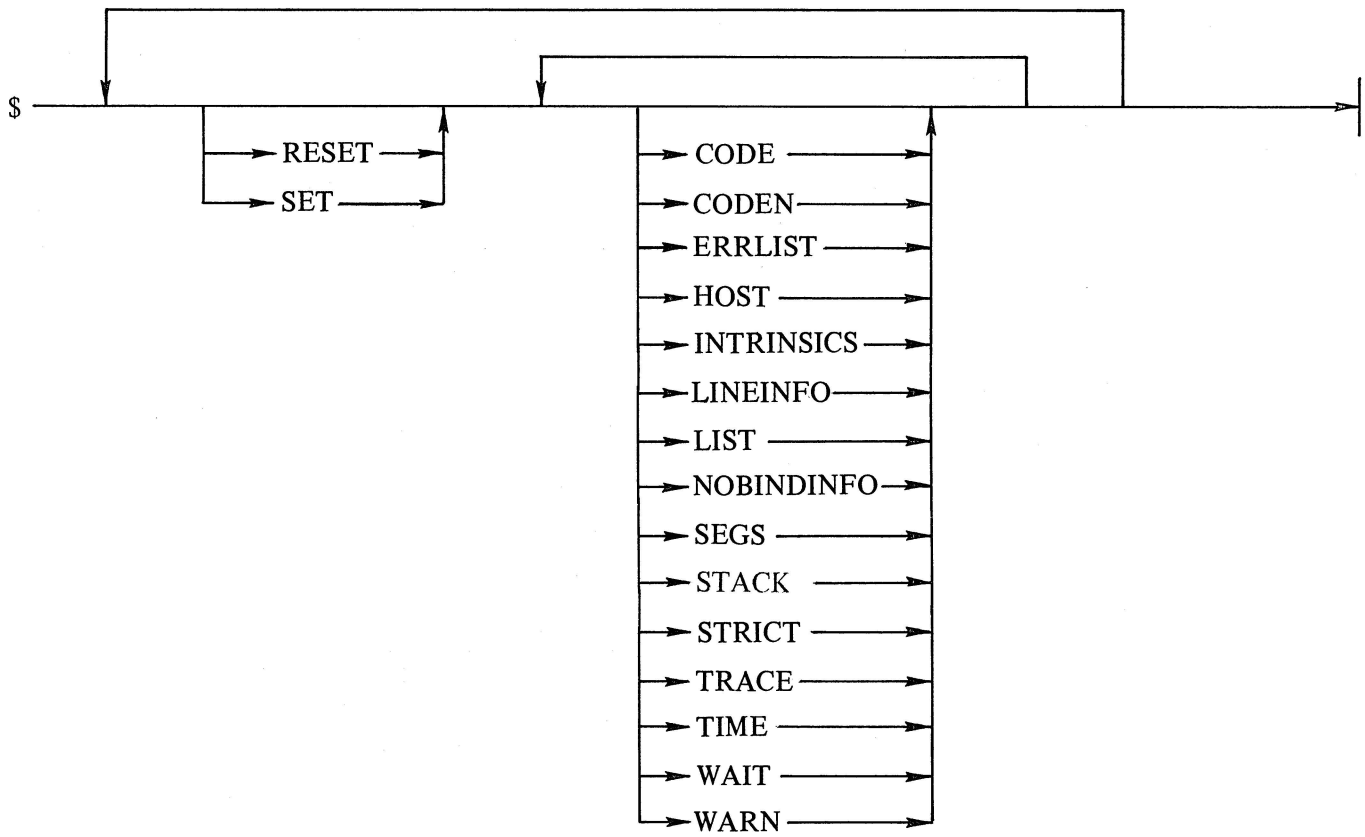
An option control card is identified by the appearance of a dollar sign ($) in the first column of the card. Binder control information is punched in the succeeding columns through column 72. As illustrated by the option control card syntax (presented on the following page), two constructs are available to the user. Option 1 allows the user to specify options that are effective throughout the binding process. Option 2 allows the user to SET certain options that are effective only during the binding of a specific subprogram.

The basic element of a Binder control statement is the compiler option, which can be invoked by the appearance of its name on an option control card. Two mutually exclusive states are associated with these options: SET and RESET; various compiler functions are dependent upon the states of such options. Default states are assigned to these compiler options, and the desired state of such an option can be specified on an option control card.

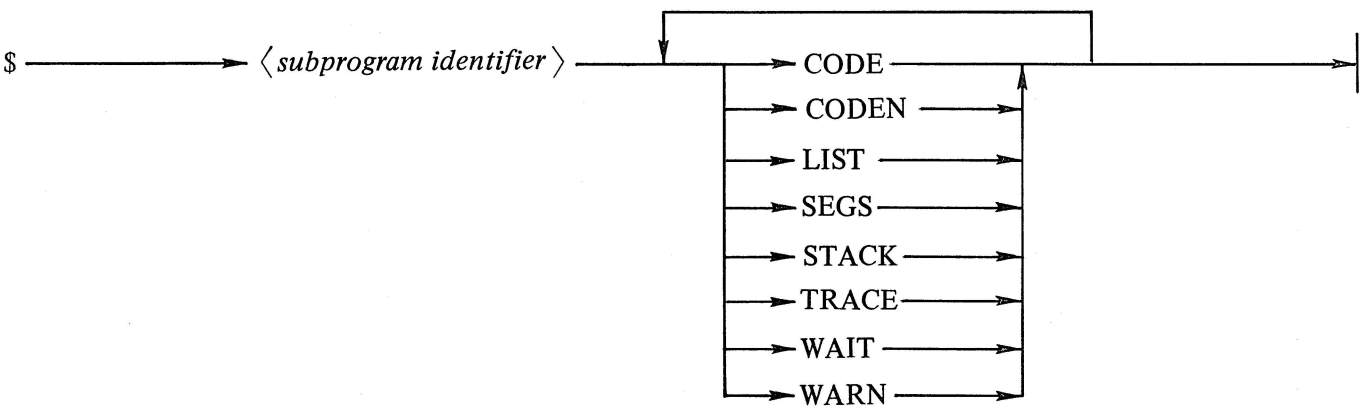
OPTION CONTROL CARDS

Syntax

OPTION 1:



OPTION 2:



Semantics

The purpose of a Binder control statement is the assignment of a desired value or state (SET or RESET) to an indicated option(s). Such a control statement must begin with a dollar sign ($) appearing in the first column of the card, followed by either an explicit or implicit option action. An explicit option action is defined as either SET or RESET.

An implicit option action is indicated when a control statement contains only a dollar sign ($) and optionally the names of options (an explicit option action is not explicitly declared). All options named in an implicit control statement are assigned the state SET, all other options are assigned the state RESET.

Options specified in a control statement by either an explicit or implicit option action are effective throughout the entire binding process. For any particular option(s), the last option card affecting that option(s) has precedence over all others.

As previously stated, certain options may be SET which are effective only during the binding of a specific procedure(s); all other specified options are ignored during this binding period. Option 2 of the option control card syntax provides a list of the options which may be SET during the binding of desired subprograms. As many specific procedure control statements as desired may be included in the input card deck; however, only one ⟨*subprogram identifier*⟩ per option card is allowed. Once the binding of the specified subprogram is completed, those options previously specified are again effective and the remainder of the binding process is performed under the control of those options.

The following statements are examples of Binder control statements employing the SET and RESET option actions and the specific procedure option specification.

```
$ SET CODE STACK LIST;
$ RESET SEGS SINGLE;
$ PROCEDURE1 CODE TRACE WAIT WARN;
```

OPTIONS

Binder options are discussed alphabetically in the following paragraphs. The default state of each option is indicated in parentheses following the option name; the function performed by the option is discussed in the paragraph following the same.

Binder options and their functions are as follows:

CODE (RESET)

Causes the output code to be printed in hexadecimal form.

CODEN (RESET)

Causes the input code to be printed in hexadecimal form.

ERRLIST (RESET; SET for CANDE)

Causes Binder to write error messages to a separate file.

HOST (SET)

Indicates that a Host file is required. HOST is RESET by default when the INTRINSICS option is SET.

INTRINSICS (RESET)

This option must be SET prior to the creating of an intrinsic file or the performing of any intrinsic binding. The setting of this option causes the HOST option to be RESET.

LINEINFO (SET)

Directs the Binder to maintain all LINEINFO information encountered in the Host and subprogram files. If RESET, all LINEINFO information encountered in the Host and subprogram files is purged from the resultant code file.

LIST (SET; RESET for CANDE)

Causes all input cards, if any, to be listed on the line printer. LIST also causes all ⟨*identifier*⟩s to be listed. Those ⟨*identifier*⟩s whose location is changed are also listed.

NOBINDINFO (RESET)

Causes the Binder to purge all Binder information from the resultant program. The resultant program may not be used as input to the Binder if the Binder information has been purged.

SEGS (SET)

Causes the listing of all segment dictionary changes. SEGS is SET by default when LIST is SET; SEGS is RESET by default when LIST is RESET.

STACK (RESET)

Causes the listing of address couples of all ⟨*identifier*⟩s encountered.

STRICT (RESET; SET for MCP)

Keeps a code file from being locked if for any reason a subprogram specified in a BIND statement is not bound.

TRACE (RESET)

Causes the printing of the following for debugging of the Binder:

a. Enter or exit procedure names of selected procedures within the Binder.

b. Selected arrays such as INFOTABLE at selected times.

TIME (RESET)

Causes the printing of header and timing information if it is not otherwise printed.

WAIT (RESET)

Causes the Binder to suspend the binding process if a specified subprogram file is not present on disk. This option allows the operator to make the file present, to terminate the Binder, or to enter the OF or FA operator command. Any OF or FA operator command applies to that file only. Subsequent non-present files cause a NO FILE condition.

WARN (SET)

Causes a message to be printed for each condition which might be an error as opposed to a definite error condition. WARN is SET by default when LIST is SET; WARN is RESET by default when LIST is RESET.

Appendix C. BINDER FILES

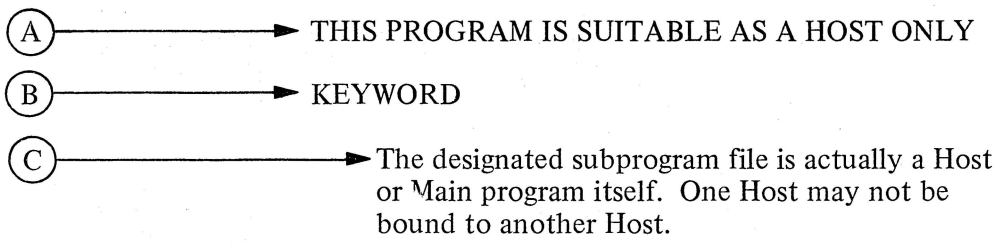
Table C-1, Binder Files Table, lists the internal name of all Binder files, the purpose served by each file, the default KIND of the file, the code used to store file data, the default record size (MAXRECSIZE) and block size (BLOCKSIZE) of the file, and a brief commentary on the file.

Table C-1. Binder Files Table

INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE (in Words)	BLOCK SIZE	COMMENTS
CARD	Input Card File	CARD READER	EBCDIC BCL	14 10	Blocked or unblocked	Optional input file directs the Binder if required.
CODE	Bound Object-Code File	DISK	Hexadecimal	30	600 words	Resultant object code file. Saved or discarded and assigned a title as indicated by the BIND control card. The default file title after a bind is the name specified in the ⟨*bind statement*⟩.
ERROR-FILE	Error Listing Output File	LINE PRINTER	EBCDIC	22	22 words	Optional error listing file produced when ERROR-LIST Binder option is SET. Automatically provided for CANDE input.
HOST	Host File	DISK	Hexadecimal	30	600 words	Contains the Host object code.
LINE	Line Printer	LINE	EBCDIC	22	22 words	Optional and label-equatable file. Produced when LIST is SET.

Appendix D. ERROR MESSAGES

This appendix contains a complete list of the error messages that can be generated by the Binder. All error messages are listed in alphabetical order. The following is an explanation of the format used to define an error message and a definition of the indicated (circled) items is given below.



Item	Definition
(A)	Error message that appears on listing.
(B)	Binder procedure name. Multiple procedure names indicate that the error message can be generated by several procedures.
(C)	Explanation of error message.

BINDER ERROR MESSAGES

ATTEMPT TO ADD GLOBAL STACK CELL TO HOST WITH NO GLOBAL
BUILDADRCPL

The Host is a subprogram which contains no global declarations. The Binder will not allow a new global to be added to such a Host in the course of binding a nested subprogram.

ATTEMPT TO ADD NEW GLOBAL TO INTRINSIC FILE WITHOUT BIND CARD
ENTEROBEXT
KEYWORD

An intrinsic being bound to an intrinsic file references a global variable which (1) is not an MCP global item, (2) is not initialized to a correct address couple via an ⟨*initialize statement*⟩, (3) does not already exist in the intrinsic file, (4) and is not specified to be bound on a ⟨*bind statement*⟩.

BINDER ERROR – PROCDRI AND INFO MISMATCH
BIND

(Refer problem to technical support.)

COMMA EXPECTED
PFILENAMES

COMPILED WITH INCOMPATIBLE VERSION
READSEGDICT

Applies only to ESPOL/MCP binding. When the MCP is compiled, the ESPOL compiler puts the version in the Info file which is created. Subprograms which are compiled using the MCP Info File are marked with the version of the Info file. When binding the subprogram into the MCP, the Binder checks to make sure the version of the MCP is compatible with the version of the Info File used in compiling the subprogram.

COMPILER ERROR
KEYWORD

(Refer problem to technical support.)

CONSTANT ARRAY OVERFLOW – INCREASE CONSARRAYLEN
ADDCONSTANT
ENTERCONSTANT

(Refer problem to technical support.)

DUE TO THE ABOVE ERROR(S) BINDING OF THIS PROCEDURE IS DISCONTINUED (CODE FILE MAY STILL BE LOCKED)
PREPARETOBACKOUT

The definition of a subprogram within a subprogram file has been found to be incompatible with the subprogram's definition in the Host. The reasons for the incompatibility should be indicated by error messages emitted just prior to this message. The Binder discontinues binding the procedure at this point, resets the error count back to the value it has previous to 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 are known by the same ⟨*identifier*⟩ and the Binder attempts to bind both occurrences of the ⟨*identifier*⟩ from the same subprogram file, the definition of the separate subprogram probably would be incompatible with one of the occurrences but might be compatible with the other occurence. Thus the subprogram would be bound correctly to its compatible occurrence and not affect the incompatible occurrence in an adverse way. This might have been the original intention of the user who did not realize that a ⟨*subprogram identifier*⟩ occurred twice within the Host.

DOLLAR OPTION ILLEGAL IN MIDDLE OF STATEMENT
DOLLAROPTION

Binder option cards may appear between input cards but may not appear between statements contained on more than one card.

ENTRY POINT CANNOT BE ADDED AT OTHER THAN GLOBAL LEVEL
KEYWORD

If a FORTRAN subprogram containing entry points is compiled at a level higher than 3 and bound to a Host 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 (LEVEL 2) which is incompatible with its execution level. When binding a higher level subprogram containing entry points, all entry points must be declared directly within the program unit to which the subprogram is bound.

EQUAL EXPECTED
PFILENAMES

ESPOL MAY BE BOUND TO ESPOL ONLY
READSEGDICT

FOR EXPECTED
PFILENAMES

HOST COMPILED AT TOO HIGH LEVEL
KEYWORD

The Host was not compiled at level 2.

HOST FILE IS NOT INTRINSIC ($ INTRINSIC SET)
READSEGDICT

The INTRINSICS option is SET in the Binder, but the Host is not an intrinsics file.

IDENTIFIERS OF SEPARATE PROCEDURE AND HOST DIFFER
OUTMESS

The Binder was directed to bind the given subprogram from a specific file via a ⟨*bind statement*⟩. The ⟨*subprogram identifier*⟩ in the subprogram file does not match the declaration in the Host. The Binder emits this warning and creates a ⟨*use statement*⟩ which matches the two ⟨*identifier*⟩s. Note that this situation cannot occur when binding from a specific library file or multiprocedure file.

ILLEGAL FILE NAME
BUILDFILENAMES

ILLEGAL IDENTIFIER
DOLLAROPTION
PFILENAMES
PROCESSQUALIFIERS

INCOMPATIBILITY IN LOWER BOUNDS
CHECKINFO

When an array is passed as a parameter to a subprogram, the array itself may be passed (or the array plus an offset (LOWER BOUNDS) may be passed) according to the specified language. FORTRAN always passes an array descriptor plus an offset. COBOL rarely passes an offset although it will accept or pass an offset if the "WITH LOWER BOUNDS" clause is used: however, the offset value itself will be ignored in calculating subscripts within the COBOL subprogram. In ALGOL the user may specify whether or not the array parameter is passed with LOWER BOUNDS. This error message results when the Binder detects that a subprogram expects LOWER BOUNDS for an array and is not passed them, or it does not expect LOWER BOUNDS and is called with LOWER BOUNDS being passed to it.

INCORRECT INTRINSIC NUMBER – ANOTHER INTRINSIC HAS SAME NO
ENTERINTRBINDCARDS

Two intrinsics within the same intrinsic file may not have the same Intrinsic Number Pair.

INCORRECT INTRINSIC NUMBER – DUPLICATE ID EXISTS WITH DIFFERENT NO
ENTERINTRBINDCARDS

An intrinsic with the same ⟨*identifier*⟩ already exists within the intrinsic file. The existing intrinsic has a different Intrinsic Number Pair than the number pair specified for the given intrinsic which is being bound.

INCORRECT NUMBER OF ADDRESS COUPLES
KEYWORD

A PL/I structure of other variable type has a number of address couples associated with it, in accordance with the way it is declared within 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.

INCORRECT NUMBER OF DIMENSIONS
CHECKINFO

The number of dimensions for the given array used in a subprogram differs from its declaration in the Host.

INCORRECT NUMBER OF PARAMETERS
CHECKINFO
CHECKSTDINTR
KEYWORD

A subprogram was called or declared with the wrong number of parameters.

INFO TABLE OVERFLOW – BINDER CAPACITY EXCEEDED
ENTERNEWGLB
ENTERINFO

(Refer problem to Technical Support.)

INITIALIZE LEGAL FOR INTRINSIC OR MCP BINDING ONLY
KEYWORD

The ⟨*initialize statement*⟩ is legal only for intrinsic or MCP binding.

INPUT CARD CONCERNING THIS PROCEDURE WAS NOT USED - - -
BUILDPROGDESC

If the Binder is directed to bind a subprogram according to a ⟨*bind statement*⟩, but the ⟨*subprogram identifier*⟩ specified on the ⟨*bind statement*⟩ is not declared in the Host or otherwise encountered during the binding process, the subprogram is not bound.

INTEGER EXPECTED
CONVINT
PFILENAMES

IS EXPECTED
READATA

LANGUAGE IDENTIFIER EXPECTED
READATA

MISSING LEFT PAREN
PFILENAMES

MISSPELLED OPTION
DOLLAROPTION

NO INTRINSIC NUMBER GIVEN – CANNOT BE REFERENCED OUTSIDE INTRINSICS
ENTERINTRBINDCARDS

This warning is given when a new intrinsic is added to an intrinsic file without an Intrinsic Number Pair being specified for it on the ⟨*bind statement*⟩. The intrinsic may be called by other intrinsics within the file but may not be invoked from a user program.

NOTHING BOUND
BIND

The Binder detected that no subporgrams were bound to the Host during the binding process.

NUMBER OF FILES TOO GREAT – INCREASE NUMFILES
FINDFILE

The number of file declarations which the Binder has reserved for subprogram files has been exceeded. Each time the Binder recurses a level in order to bind a nested external subprogram, an additional subprogram file declaration is required. In addition each library or multiprocedure file which is opened by the 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 will result. Refer problem to technical support.

<- - - NEW GLOBAL ADDED TO HOST
KEYWORD

In attempting to match a reference in a subprogram to its definition in the Host, the Binder discovered that the particular variable did not exist in the Host. The variable is then added to the Host at the global level and this warning is given.

OFFSET OF XXXX CANNOT BE REACHED FROM LEVEL XX
CHECKOFFSET

As the execution level of a subprogram increases, the offset which may be specified in a VALC or NAMC operator decreases.

ONLY MULTIPROCEDURE FILES ARE ALLOWED IN UNIVERSAL BIND STATEMENT
BINDFILE

One of the following two types of statements was given as input to the Binder:

```
BIND = FROM A/B
BIND P, Q, SUBR FROM A/B;
```

where A/B is not a library or multiprocedure file. Since A/B contains only one subprogram it should not be used in the above ⟨*bind statement*⟩.

PL/I MAY BE BOUND TO PL/I ONLY
READSEGDICT

PROCEDURE PASSED AS PARAMETER NOT DECLARED FORMALLY
CHECKFURTHER
KEYWORD

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 callee 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.

QUOTE EXPECTED
SKAN

RECOMPILE WITH MARK 2 OR HIGHER
READSEGDICT

The code file was compiled with a compiler version prior to II.0.

RIGHT PAREN EXPECTED
PFILENAMES

SAVE CODE CANT ADD GLOBALS
BINDSAVECODE
SCANSAVSD

While binding MCP segment 1, new globals may not be added to the MCP.

SEMICOLON EXPECTED
PFILENAMES

SUBPROGRAM SPECIFIER EXPECTED
PFILENAMES

THIS FILE HAS NO BINDER INFORMATION
READSEGDICT

The file is not suitable as input to the Binder. The option "NOBINDINFO" may have been set during the file's creation or the respective compiler may have determined that the file contained no external references and therefore did not require binding.

THIS FILE NOT CODE FILE
FINDFILE
OUTER BLOCK OF BINDER

Check to make sure the file title is complete and that a directory is not specified by the title.

THIS FILE NOT PRESENT
BINDSAVECODE
BINDFILE
FIXOBCODE

THIS FILE WAS PREVIOUSLY BOUND – SUITABLE AS HOST ONLY
READSEGDICT

The resultant code file from a previous bind may not be bound to another Host. Such a file may only be used as a Host in subsequent binds.

THIS PROCEDURE CANT BE PASSED BETWEEN THESE LANGUAGES
CHECKINFO

A procedure may not be passed as a parameter between the two languages.

THIS PROCEDURE NOT FOUND IN MULTIPROCEDURE FILE(S)
FINDFILE

The Binder was not able to find the subprogram within the library files designated in the ⟨*bind statement*⟩. The subprogram is left in the Host in its present form, and binding of other subprograms continues.

THIS PROGRAM IS SUITABLE AS A HOST ONLY
KEYWORD

The designated subprogram file is actually a Host or Main program itself. One Host may not be bound to another Host.

THIS PROGRAM UNIT COMPILED AT INCOMPATIBLE LEVEL
($ SET LEVEL N)
KEYWORD

The given subprogram was compiled at a level incompatible with its execution level within the Host. Recompile the subprogram with the correct execution level set via the compiler's LEVEL option.

THIS VARIABLE TYPE CANNOT BE ADDED TO HOST
ENTEROBEXT

In attempting to match a reference in a subprogram to its definition in the Host, the Binder discovered that the particular variable did not exist in the Host. Normally the Binder would add the variable to the Host, but the Binder is incapable of adding the following variable types to a Host:

FORMAT
PICTURE
LABEL
"SWITCH TYPE ITEMS"
LIST

TOO MANY ENTRIES – INCREASE MAXENTRIES
ENTEREFS

(Refer problem to technical support.)

TOO MANY QUALIFIERS
PROCESSQUALIFIERS

UNRECOGNIZED STATEMENT
PFILENAMES

UNSPECIFIED LENGTH FOR NEW ARRAY
BUILDDESCRIPTOR
EMITLENGTH

The array which was added as a new global to the Host had no length specified for it. In ALGOL or ESPOL this results from not having declared an UPPER BOUND for the array within the brackets used for declaring such globals for separate compilation. In COBOL this message occurs any time a new array is added to a Host by the Binder. (New global arrays are therefore not allowed for COBOL binding.)

INDEX