Binder Manual

Abstract

This manual describes Binder, the product that reads, links, and produces object files. This manual is a user's guide and reference manual for software developers who need to link and modify object files.

Product Version

T9621 at H06.03 and D30

Supported Release Version Updates (RVUs)

This manual supports D30.00 and all subsequent D-series RVUs, G02.00 and all subsequent G-series RVUs, and H06.03 and all subsequent H-series RVUs until otherwise indicated in a new edition.

Part Number	Published
528613-003	July 2005

Document History

Part Number	Product Version	Published
48509	BINDER C30	September 1991
085670	BINDER D10	January 1993
109641	BINDER D30	December 1994
528613-001	BINDER D30	August 2004
528613-003	T9621 H06.03	July 2005

Binder Manual

Glossary | Index | Figures | Tables

What's New in This Manual vii
- Manual Information vii
- New and Changed Information vii

About This Manual ix
- Notation Conventions ix

1. Introduction

Definition of binding 1-1
Forms of Binder 1-2
- BINSERV 1-2
- BIND 1-3

Languages Used with Binder 1-4
Relation of Binder to Crossref, Inspect, and the Accelerator 1-5

2. Using Binder

Running Binder 2-1
- Manual Operation 2-1

Command File Operation 2-2
- Examples 2-2

Defining the Target File 2-3
- Specifying Input File Names 2-3
- Specifying the Target File Order 2-4

Binding Modules 2-5
- Binding Rules 2-6
- Examples 2-9
- Binding COBOL85 and FORTRAN Programs 2-10
- Binding C Programs 2-11
- Binding Pascal Programs 2-12
- Binding Mixed-Language Programs 2-13
- Parameter Checking for a Mixed-Language Bind 2-14
- Binding SQL Program Files 2-14
- Resolving External References 2-16

2. Using Binder (continued)

Specifying a Different Volume for Binder Work Files 2-17
- When Using BIND 2-18
- When Using BINSERV 2-18
- Specifying the Swap Volume for Pascal and C Programs 2-18

Generating Output Listings 2-19
- Target File Statistics 2-19
- Load Maps 2-21

Cross-Reference Lists 2-25

3. BIND Commands

Summary of Most Commonly Used Commands 3-2
How to Use BIND Commands Efficiently 3-3
- Searching for Files 3-3
- Replacing Procedures 3-3
- Turning off Load Maps 3-4
- Binding without Fixups 3-4

Syntax Conventions for Name Lists as Command Elements 3-4
ADD Command 3-6
ALTER Command 3-9
BUILD Command 3-10
CD Command 3-14
CHANGE Command 3-15
CLEAR Command 3-18
COMMENT Command 3-18
DELETE Command 3-19
DUMP Command 3-20
ENV Command 3-22
EXIT Command 3-22
FC Command 3-22
FILE Command 3-23
HELP Command 3-23
INFO Command 3-24
LIST Command 3-27
LMAP Command 3-31
LOG Command 3-31
MODE Command 3-32
MODIFY Command 3-33
MOVE Command 3-35

3. BIND Commands (continued)

OBEY Command 3-37
OUT Command 3-38
RENAME Command 3-39
REPLACE Command 3-40
RESELECT Command 3-42
RESET Command 3-44
SATISFY Command 3-46
SELECT Command 3-48
SET Command 3-56
SHOW Command 3-65
STRIP Command 3-73
SYSTEM Command 3-75
VERIFY Command 3-75
VOLUME Command 3-76

4. Object File Structure

Code Blocks, Entry Points, and Data Blocks 4-1
Code Blocks 4-1
Primary and Secondary Entry Points 4-2
Data Blocks 4-4
Object File Format 4-8
Header 4-9
Code Region 4-9
Data Region 4-11
Accelerator Region 4-11
Inspect Region 4-11
Binder Region 4-11

5. Binder Input and Output

The Input Control Lists 5-1
Creating the Input Control Lists 5-2
How Binder Uses the Input Control Lists 5-3
The Target File 5-8
Target File Attributes 5-8
How Binder Builds the Target File 5-11

6. User Libraries

Binding User-Library Procedures 6-1
Object File Format 6-2
Preventing Binder Resolution of Library Calls 6-2
    Compilation-Time Binding 6-2
    Command-Driven Binding 6-2
Specifying a User Library 6-3
Restrictions on User Libraries 6-3
Shared Run-Time Libraries 6-4
    Building Applications That Use SRLs 6-4
    Using Binder Commands With SRLs 6-4
    Reserving Space With the SET RESERVE Command 6-5

7. Guardian File Names and TACL Commands

Disk File Names 7-1
    Parts of a Disk File Name 7-2
    Partial File Names 7-3
    Logical File Names 7-4
    Internal File Names 7-4
TACL Commands 7-4
TACL DEFINE Commands 7-5
    Substituting a File Name 7-5
    TACL DEFINE Names 7-5
    Setting DEFINE CLASS Attributes 7-6
TACL PARAM Commands 7-7
    PARAM BINSERV Command 7-7
    PARAM SAMECPU Command 7-7
    PARAM SWAPVOL Command 7-8
    PARAM SYMSERV Command 7-8
    Using PARAM Commands 7-8
TACL ASSIGN Commands 7-9

8. Binder Messages

Error Messages and Warnings 8-1
Completion Codes 8-28

9. Syntax Summary

Glossary

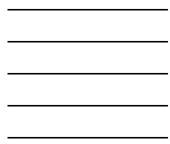
Index

Figures

Figure 1-1. BINSERV: Compilation-Time Binding 1-3
Figure 1-2. BIND: Command-Driven Binding 1-4
Figure 2-1. External References 2-17
Figure 2-2. Target File Statistics 2-20
Figure 2-3. Alphabetic Entry Point Map for Entry Points and Code Blocks 2-22
Figure 2-4. Entry Point Map by Location for Multiple Code Segments 2-23
Figure 2-5. Alphabetical Load Maps for Data Blocks 2-24
Figure 2-6. Listing for Read-Only Data Blocks 2-25
Figure 2-7. Cross-Reference Listing 2-26
Figure 4-1. Example of the Binder Object File Format 4-8

Tables

Table 1-1. Languages used with Binder 1-4
Table 2-1. Target File Specifications built by Binder 2-4
Table 2-2. Binder Grouping of ENV Directive Parameters 2-7
Table 2-3. Run-Time Environment Resulting From Binding Modules 2-9
Table 2-4. Commands that Produce Listings 2-19
Table 2-5. Binder Statistics 2-20
Table 2-6. Information Included in Load Maps for Entry Points 2-22
Table 2-7. Information Included in Load Maps for Data Blocks 2-24
Table 3-1. Commonly Used BIND Commands 3-2
Table 3-2. Syntax Conventions for Named Lists 3-4
Table 3-3. Resulting Target Processor Type 3-62
Table 4-1. Code Block Attributes 4-2
Table 5-1. Commands That Create Control Lists 5-2
Table 5-2. Target File Attributes 5-9
Table 6-1. Binder Commands Used With Shared Run-Time Libraries 6-5
Table 8-1. Binder Completion Codes 8-28
Table 9-1. Binder Command Summary 9-1



What's New in This Manual

Manual Information

Abstract

This manual describes Binder, the product that reads, links, and produces object files. This manual is a user's guide and reference manual for software developers who need to link and modify object files.

Product Version

T9621 at H06.03 and D30

Supported Release Version Updates (RVUs)

This manual supports D30.00 and all subsequent D-series RVUs, G02.00 and all subsequent G-series RVUs, and H06.03 and all subsequent H-series RVUs until otherwise indicated in a new edition.

Part Number	Published
528613-003	July 2005

Document History

Part Number	Product Version	Published
48509	BINDER C30	September 1991
085670	BINDER D10	January 1993
109641	BINDER D30	December 1994
528613-001	BINDER D30	August 2004
528613-003	T9621 H06.03	July 2005

New and Changed Information

This revision describes the changes and additions made to the D30 product version of Binder (T9621) and the *Binder Manual.* This revision also incorporates support for T9621 on TNS/E systems running H06.01 software.

Summary of Changes

This edition of the manual includes previously undocumented warnings and messages for the Binder product. The changes include:

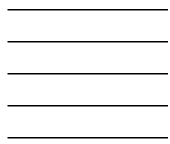
- This publication has been updated to reflect new product names.

- Because the product names are changing over time, this publication might contain both HP and Compaq product names.
- Product names in graphic representations are consistent with the current product interface.
- Section 8, Binder Messages now includes the descriptions of error 770, fatal error 233, and warning 234. The message text for error 15 and warning 106 has now been updated to match the current product interface.
- The description of the USERLIBRARY attribute is changed in Section 3, BIND Commands.

Section One, the Introduction, has been extended to include an introduction to The Object Code Accelerator (OCA) on page 1-6.

The following commands have been enhanced to support TNS/E usage:

- CHANGE Command on page 3-15
- RESET Command on page 3-44
- SET Command on page 3-57
- SHOW Command on page 3-67
- STRIP Command on page 3-75.



About This Manual

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

file-name

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

pathname

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
```

```
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on

each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]
```

```
K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }
```

```
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]...
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
```

```
LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                   !i
                         , error        ) ;             !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                     !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length          !i:i
                            , filename2:length ) ;      !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                       !i
                        , [ filename:maxlen ] ) ;       !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE

?123

CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

Event number = *number* [Subject = *first-subject-value*]

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

proc-name trapped [in SQL | in SQL file system]

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

obj-type obj-name state changed to *state*, caused by
{ Object | Operator | Service }

process-name State changed from *old-objstate* to *objstate*
{ Operator Request. }
{ Unknown.   }

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

Transfer status: { OK | Failed }

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

%005400

%B101111

%H2F

P=%*p-register* E=%*e-register*

UPPERCASE LETTERS. Uppercase letters indicate names from definition files. Type these names exactly as shown. For example:

ZCOM-TKN-SUBJ-SERV

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Introduction

The Binder program development tool reads, links, modifies, and produces executable object files for the C, COBOL85, FORTRAN, and TAL compilers. Binder also operates as an independent process for C, COBOL85, FORTRAN, TAL, and Pascal object files. This manual focuses on the use of Binder as an independent process.

For a description of the use of Binder with language compilers, see the appropriate language programmer's guide or reference manual. This introduction addresses the following topics:

Topic	Page
Definition of binding	1-1
Forms of Binder	1-2
Languages Used with Binder	1-4
Relation of Binder to Crossref, Inspect, and the Accelerator	1-5
The Object Code Accelerator (OCA)	1-6

Definition of binding

Binding is the operation of examining, collecting, linking, and modifying code and data blocks from one or more object files to produce a target object file. The input files to Binder and the output files created by Binder are all object files. To help you to distinguish between the two types of files, this manual refers to an output file as a target file.

All Binder operations are performed on object files. This manual uses the following terms when discussing object files:

Block	The smallest unit of code or data that can be relocated as a single entity. You can compile data separately in FORTRAN as COMMON and in TAL as BLOCK structures.
Object file	One or more code and data blocks compiled and bound together.
Program	An executable object file. It must contain an entry point with the MAIN attribute.

```
BINDER - OBJECT FILE BINDER - T9621D30 - (31OCT94) SYSTEM \COCOLAT
Copyright Tandem Computers Incorporated 1982-1994
```

Each compiler has its own conventions for specifying block names for code and data. Section 4, Object File Structure gives comparative information for source language constructs and the resulting blocks. For additional information, consult the language reference manual for the language that you are using.

Forms of Binder

Binder has two forms: BINSERV and BIND.

- BINSERV is the form of Binder that a language compiler uses. BINSERV builds executable object files for C, COBOL85, FORTRAN, and TAL. (The Pascal compiler uses BINSERV but does not produce an executable object file.)
- BIND is the form of Binder you can use interactively to modify object files. You can use BIND to update and link object files from C, COBOL85, FORTRAN, Pascal, and TAL. BIND is the only form of Binder that produces executable object files for Pascal programs.

BINSERV

BINSERV is a process that builds object files for a compiler process. It executes as a separate process during a compilation. BINSERV accepts commands from the compiler, builds lists of references that must be resolved, reports on its success in locating needed blocks, and eventually creates a target object file or reports its failure to do so. BINSERV also returns information to the compiler about the characteristics of the blocks it found.

BINSERV can link program units written in different languages to form an application program. For example, a server in a main COBOL85 program can use FORTRAN subprograms for calculations and TAL procedures for block moves and scans of data. For additional information on mixed-language binding, see Binding Mixed-Language Programs on page 2-13.

All object files that serve as input to Binder originate either from the compilation-time process BINSERV or from previous BIND sessions. The new object files produced can then serve as input to further binding operations. The target object file BINSERV creates has the standard object file format.

If you do not need special binding operations, you can simply compile and execute programs. In the following example, the FORTRAN compiler automatically invokes BINSERV, and the binding step is transparent to you:

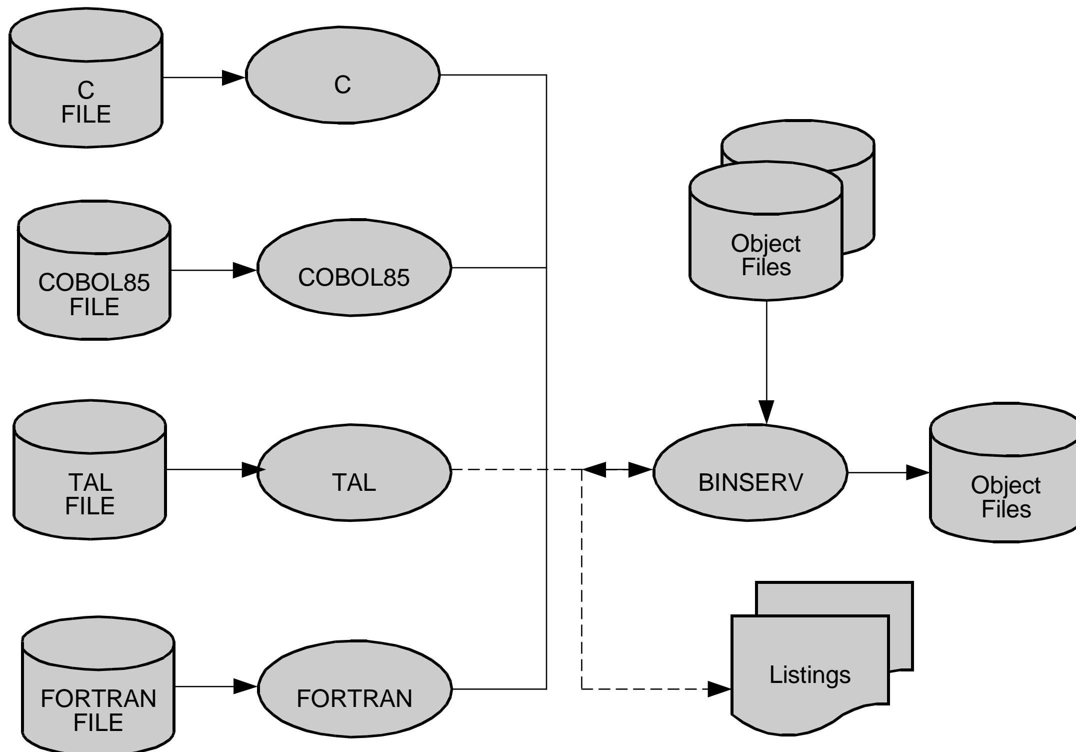
```
12> FORTRAN/in source, out list/try1
13> RUN try1
```

Figure 1-1 shows how BINSERV uses information from the compilers to produce an executable object file.

Pascal does not allow binding at compile time. For more information, see Binding Pascal Programs on page 2-12.

.

Figure 1-1. BINSERV: Compilation-Time Binding



BIND

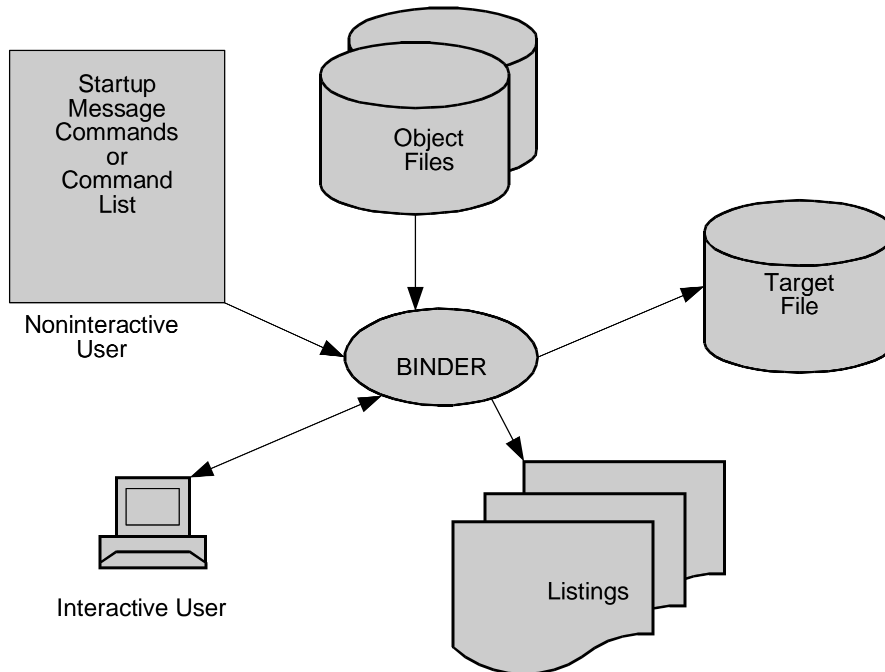
BIND is the interactive form of Binder you can use to examine, modify, and link object files. You can give BIND its commands from a terminal or a command file. In either way, you can perform any of the following functions with Binder:

- Build a target file from separate object files.
- Build a target file for use as a library of shareable procedures.
- Display object file contents.
- Modify the values of words in the code and data blocks of the target file.
- Request consistency checks for parameters or for common data blocks.
- Reorder code blocks in a target file.
- Specify external references you do not want Binder to resolve.
- Produce optional load maps and cross-reference listings.

- Reduce the size of object files by stripping them of the Binder region and Inspect region (symbol tables).
- Specify a user run-time library.

Figure 1-2 illustrates Binder operation in command-driven mode.

Figure 1-2. BIND: Command-Driven Binding



Languages Used with Binder

Table 1-1 summarizes the languages and their compiler versions that can be used with BINSERV and BIND.

Table 1-1. Languages used with Binder (page 1 of 2)

Language	BINSERV	BIND
C*	C20 or later	C00 or later
COBOL85	C20 or later	B30 or later
FORTRAN	C20 or later	E00 or later

Table 1-1. Languages used with Binder (page 2 of 2)

Language	BINSERV	BIND
Pascal*	C20 or later	B30 or later
TAL	C20 or later	E01 or later

*BINSERV builds executable object files automatically for C00 or later versions of C when a specific directive is used; it does not produce executable object files for the Pascal compiler. For more information, see Binding C Programs on page 2-11 or Binding Pascal Programs on page 2-12

Relation of Binder to Crossref, Inspect, and the Accelerator

The Binder, Inspect, and Crossref tools are designed to complement each other in the program development cycle. The Accelerator is designed for use after you have written and debugged your program.

The Inspect symbolic debugger enables you to debug a program or multiple programs symbolically. That is, you can specify debugging commands using the same identifiers you used in the source code. Because the Inspect debugger must find symbol tables in the object module to allow this feature, the compilers provide the symbol information to Binder at compile time. For more information on the Inspect product, see the *Inspect Manual*.

Crossref, a debugging tool, provides symbol reference information in the detail that you specify. Crossref listings contain source code symbols while Binder cross-reference information is for object files (entry points and common data blocks only). See the *Crossref Manual* for more information.

The Accelerator processes a TNS object file for use on either a TNS or a TNS/R processor. The Object Code Accelerator processes a TNS object file for use on a TNS/E processor.

Binder contains several command options that affect accelerated object code. They are included in the CHANGE, RESET, SHOW, and STRIP commands. These command options perform functions such as defining which processor the code can run on. For more information about these commands, see Section 3, BIND Commands.

For more information about the Accelerator, see the *Accelerator Manual*.

Note. You can submit files bound with previous versions of the Binder to the Accelerator. The special command options contained in this version of Binder are not required.

The Object Code Accelerator (OCA)

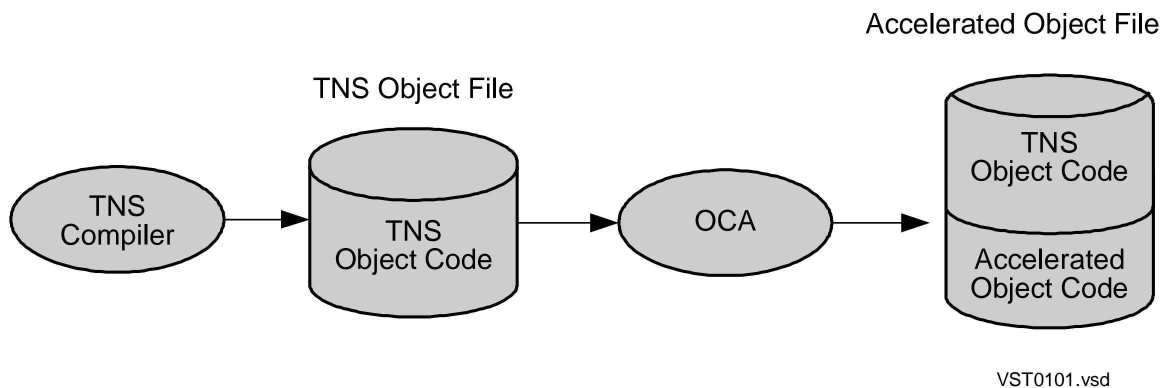
This section contains a general introduction to OCA; for more details see the *Object Code Accelerator Manual*.

OCA processes TNS object code to produce accelerated object code. On TNS/E systems, accelerated object code runs significantly faster than TNS object code.

OCA takes as input an executable TNS object file and produces as output an accelerated object file. The accelerated object file contains both the original TNS code, and the logically equivalent optimized Itanium instructions (accelerated object code).

This process is illustrated in Figure 1-3, The Acceleration Process.

Figure 1-3. The Acceleration Process



For each TNS instruction OCA produces its functional equivalent on the TNS/E system, in the form of either:

- A sequence of zero or more (usually more than one) Itanium instructions
- A call on a millicode routine

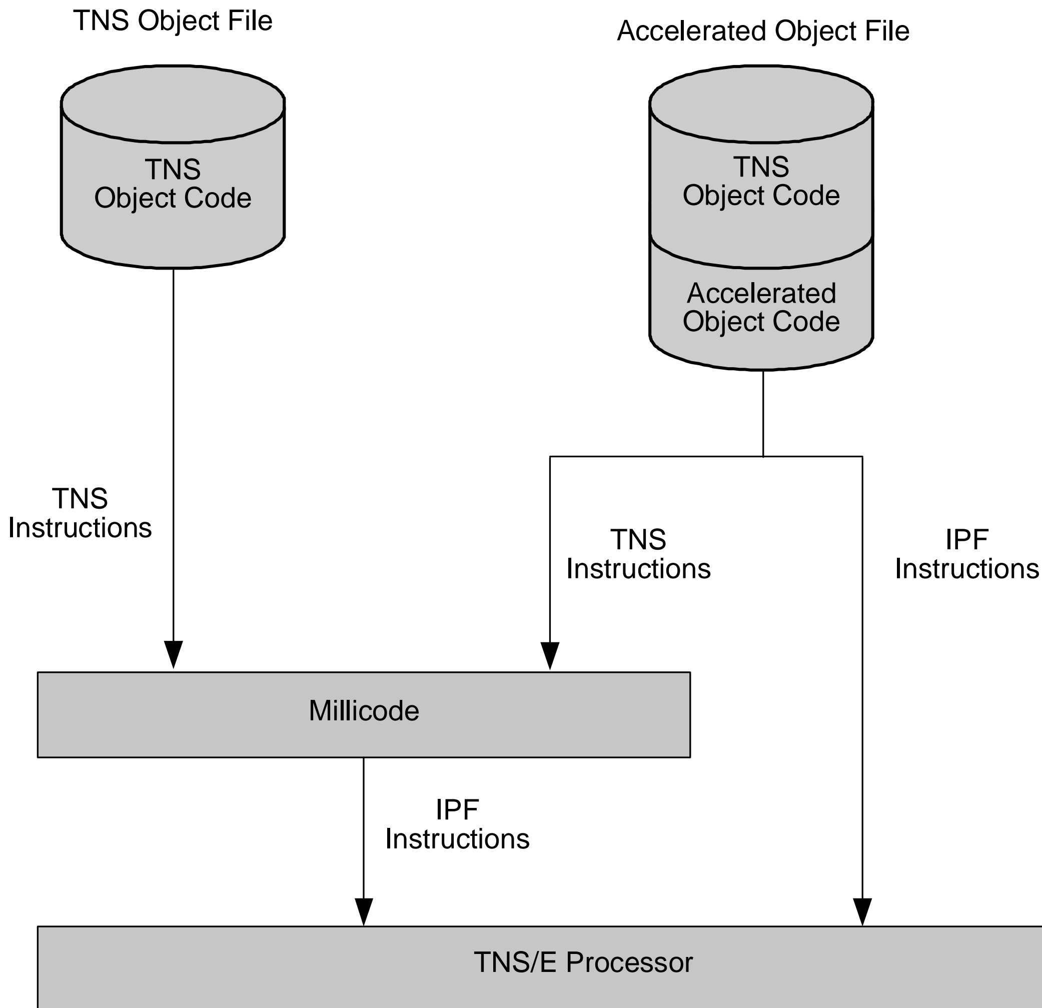
In the first and more common case, OCA treats a TNS instruction as though it were a macro instruction, expanding it into Itanium instructions. In the second case, OCA treats a TNS instruction as though it were a call on a closed subroutine, referred to as a millicode routine. Millicode routines are sets of Itanium instructions that implement various complex TNS instructions and low-level functions such as exception handling, real-time translation routines, and floating-point and quad arithmetic. Millicode is functionally equivalent to TNS microcode.

OCA can produce only optimized Itanium instructions for TNS instruction sequences whose exact meaning can be determined before runtime. When the exact meaning cannot be determined until run time, the program makes a transition into TNS code and executes the original TNS instructions.

Figure 1-4, User Program Execution Modes on TNS/E Systems shows two of the execution modes available to user programs on TNS/E systems: TNS mode and accelerated mode.

- TNS mode is the operational environment in which TNS instructions execute. On TNS/E systems, TNS instructions are implemented by millicode routines.
- By default, accelerated mode is the operational environment in which Itanium instructions generated by OCA execute. On TNS/E systems, Itanium instructions execute directly on the Itanium processor.

Figure 1-4. User Program Execution Modes on TNS/E Systems



OCA runs on TNS/R and TNS/E systems, and produces the same output in both environments: a file containing both the original TNS/R code and its Itanium code equivalent. Accelerated object files can run on both TNS/R and TNS/E systems, although TNS/R systems ignore the Itanium code.

OCA can process bound, executable object programs generated by B40 and later versions of the C, COBOL85, FORTRAN, Pascal, and TAL compilers. OCA processes executable object programs that run only in the Guardian environment. (You cannot accelerate object programs generated by the COBOL 74 compiler.)

OCA Translation Mode

On TNS/E systems, TNS objects are not run in accelerated mode by default.

You must explicitly accelerate your TNS program as part of the build or install process. If you explicitly accelerate your program in this way, you are immediately alerted of any error conditions.

If you have an accelerated object (with the corresponding IPF region) and no longer want it accelerated, use the Binder `CHANGE OCA ENABLE OFF` command to disable acceleration. See CHANGE Command on page 3-15.

Cross-Platform Acceleration

OCA is supported on D-series, G-series, and H-series systems, enabling you to accelerate TNS object files on the TNS/E platform for execution on the TNS/E platform. (A file with a TNS region only cannot be executed on the TNS/E platform). The TNS/R acceleration equivalent, Accelerator, is supported on G-series RVUs, enabling you to accelerate TNS object files on the TNS/R platform for execution on the TNS/R platform. Having both accelerators available on both platforms enables you to use either the TNS/R or TNS/E platform to create files that can be executed with improved performance on either platform.

2 Using Binder

This section describes how to:

- Run Binder
- Define the target file
- Bind modules from the same language or different languages or from NonStop SQL/MP
- Specify a volume for Binder work files
- Generate output listings

Topic	Page
Running Binder	2-1
Defining the Target File	2-3
Binding Modules	2-5
Specifying a Different Volume for Binder Work Files	2-17
Generating Output Listings	2-19
Cross-Reference Lists	2-25

Running Binder

You can run Binder directly from the operating system command interpreter, called TACL, or through a command file.

This section describes both ways of running Binder. Command keywords are shown in uppercase characters, but Binder accepts lowercase characters as well for all commands.

Manual Operation

You can manually start and stop Binder. To start a BIND process, type `BIND` at the TACL prompt as shown:

```
1> BIND
```

Binder displays the product banner and the Binder prompt `@` (an at sign) at your terminal:

```
BINDER - OBJECT FILE BINDER - T9621D30 - (31OCT94) SYSTEM \GREEN
Copyright Tandem Computers Incorporated 1982-1994
@
```

You can communicate with the BIND process by typing a command at each Binder prompt.

To stop a BIND process, type `EXIT` at the Binder prompt:

```
@EXIT
2>
```

This returns you to TACL. You can press CTRL/Y instead of typing the `EXIT` command.

For examples of and details on commands, see Section 3, BIND Commands.

Command File Operation

When you manually start a BIND process, you can specify an optional command file, an optional listing file, or both. The command file controls the process so you do not need to enter commands interactively. The listing file receives all listing output from the BIND process.

The syntax for specifying the command file and listing file in the BIND command is:

```
BIND [ / IN command-file / ]
[ / OUT listing-file / ]
[ / IN command-file, OUT listing-file / ]
[ command [ ; command ] ... ]
```

`IN` *`command-file`*

specifies the file containing BIND commands. If you omit this parameter, Binder prompts the current input file of the command interpreter, normally the home terminal.

`OUT` *`listing-file`*

specifies the file to receive output listings. If you omit this parameter, output is directed to the current output file of the command interpreter, normally the home terminal.

`command`

is any BIND command. If you specify one or more commands, Binder executes those commands and terminates without opening or reading the command file.

Note. Binder complies with the TACL ASSIGN messages that override the default characteristics of the IN and OUT files, such as record length. For a description of ASSIGN messages, refer to the *Guardian Procedure Errors and Messages Manual*

Examples

This subsection provides examples of running Binder in various ways.

- In the following example, Binder reads commands from the file BINDTAL and directs its output listing to the home terminal:

```
12> BIND / IN bindtal /
```

- In the following example, Binder prompts you at the home terminal for commands. Binder sends the output listing to the LISTTAL file:

```
13> BIND / OUT listtal /
```

- In the following example, Binder executes the commands, terminates, and returns control to TACL.

```
14> BIND / IN building, OUT list/
```

  This command causes Binder to accept and execute the commands in the file named BUILDING and to direct the output (including error messages) to the file LIST. Binder terminates when it reaches an end of file or when it encounters the EXIT command in the file BUILDING.

  If you do not specify an OUT file, Binder sends output to the default OUT file; this file is normally your home terminal.

- In the following example, Binder directs its output to the LISTC file; it ignores the command file BINDC because of the appearance of BIND commands in the BIND command line:

```
15> BIND / IN bindc, OUT listc/ ADD * from prog1
```

Defining the Target File

To define the target file, you enter BIND commands that specify the order and contents of the file. You specify the contents of the target file by identifying the code blocks and data blocks that you want to include. You specify the order of the target file contents by the order in which you specify the commands.

A block is the smallest unit that can be separately relocated. Both code instructions and data are organized into blocks. Each compiler has its own conventions for specifying block names for code and data. Section 4, Object File Structure gives comparative information for source language constructs and the resulting blocks. For additional information, consult the language reference manual for the language you are using.

If you use BIND commands to modify code or data blocks, Binder makes these changes in the target file only.

Specifying Input File Names

You specify the names of input object files using the ADD and REPLACE commands. The ADD command tells Binder where to find the blocks for the include lists.

REPLACE tells Binder to exchange the specified entry for an existing one on either the code block or data block include list.

You must supply the disk file name of the correct object file before Binder accepts the name of a code or data block for an include list. The file Binder uses as the default for block retrieval is the current file. Either a block must be in the current file or you must specify the file when you add the block. You can change the current file designation as often as you want by using the FILE command.

Specifying the Target File Order

Binder assumes the order you specify is the order you prefer for the code blocks in the target file. (You cannot establish the ordering of data blocks in the target file beforehand.)

Binder builds lists of target file specifications from your commands and from input object files. You need at least one include list for the binding process; the other lists are optional.

Table 2-1. Target File Specifications built by Binder (page 1 of 2)

Name of List	Number of Lists Allowed	Description
Include Lists	4	These lists reflect the order in which Binder encounters code blocks, data blocks, entry point names, or run-time data units (RTDUs) and the order in which Binder builds the target file. There is one include list for code blocks, one for data blocks, one for entry point names, and one for RTDUs.
Omit List	1	This list is for external references to entry points that Binder is not to resolve, regardless of the names included.
Refer List	1	This list holds pairs of entry point names; the first name satisfies references to the second. For example, an entry point to a code stub (that is, a block skeleton) can satisfy references to code that is not ready.
Modify List	1	This list contains an entry for every code or data block patch.

Table 2-1. Target File Specifications built by Binder (page 2 of 2)

Name of List	Number of Lists Allowed	Description
Unresolved Reference Lists	2	There are two unresolved reference lists, one for entry points and one for data blocks. As you enter each name on an include list, Binder removes entries from the unresolved reference lists if the entries are satisfied by the new name.
Undefined List	1	This list contains data blocks referenced, but not yet defined in C or Pascal.
Search List	1	This list holds disk file names of object files to search for unresolved references. Binder searches these files in the order in which you name them. If two files contain versions of the same block, Binder uses the first block encountered.

The undefined list differs from the unresolved list in that the undefined list contains references to just data blocks whereas the unresolved list contains references to both data blocks and code blocks. While Binder does not need to resolve unresolved code references to entry points into the operating system or system code library, Binder must resolve all data references within the process before run time. Undefined data blocks have known size, but unknown initial values. C and Pascal require a data block to be defined by exactly one module that is responsible for its initialization. The undefined data list contains the names of those data blocks whose responsible module has not yet been located.

Normally, an unresolved reference list is present during some part of the session.For descriptions of these lists and how they are created, see Section 5, Binder Input and Output.

Binding Modules

You can bind modules from the same language or from different languages. For example, suppose two object files, named OBJFILEX and OBJFILEY, contain the code and data blocks needed to make up a large program. Each object file contains multiple program units or procedures, all of which are needed in the target file. For this discussion, the source language does not matter.

You can bind the separate blocks by running BIND and then entering commands at the Binder prompt. For example:

```
12> BIND
@ADD * FROM objfilex
@ADD * FROM objfiley
@BUILD trgfilez
```

The ADD * command causes all the code and data blocks from an object file to be included in the target file. Therefore, if OBJFILEX contains PROC A, PROC B, and PROC D, and OBJFILEY contains PROC C, PROC E, and PROC F, then TRGFILEZ contains all six procedures.

If more than one object file includes the same code block name or data block name, Binder retains the first one it encounters.

Binding Rules

The rules that govern binding depend on the run-time environment. Before the D-series RVUs, C, COBOL85, FORTRAN, Pascal, and TAL each had its own run-time environment. These language-specific run-time environments were different from one another and often incompatible. With the D-series RVUs, the Common Run-Time Environment (CRE) provides a shared run-time facility for C, COBOL85, Pascal, FORTRAN, and TAL. D-series versions of the C and Pascal compilers always generate programs that run in the CRE[1]. D-series versions of the COBOL85, FORTRAN, and TAL compilers generate programs that can run in either a language-specific run-time environment or the CRE, depending on the setting of the ENV directive.

Binder categorizes the ENV directive parameters into three groups: OLD, NEUTRAL, and COMMON. For the most part, these groups match the various ENV parameters provided by the language compilers.

- OLD generates code that runs only in a language-specific run-time environment.
- COMMON generates code that runs only in the CRE.
- NEUTRAL generates code that runs in either a language-specific run-time environment or the CRE.

---

[1]The C and Pascal compilers generate code that runs in either a language-specific run-time environment or the CRE if you specify the ENV LIBSPACE or ENV EMBEDDED directives. These C and Pascal routines cannot perform I/O or access the heap, and therefore are not useful in most situations.

Table 2-2. Binder Grouping of ENV Directive Parameters (page 1 of 2)

Binder Group	Language	Generated by:
OLD	C	C-series compilers
	COBOL85	C-series compilers by default
		D-series compilers by default
		D-series compilers with ENV OLD specified
	FORTRAN	C-series compilers by default
		D-series compilers by default
		D-series compilers with ENV OLD specified
	Pascal	C-series compilers by default
		C-series compilers with ENV FULL specified
	TAL	D-series compilers with ENV OLD specified
COMMON	C	D-series compilers by default
		D-series compilers with ENV COMMON specified
		D-series compilers with ENV LIBRARY specified
	COBOL85	D-series compilers with ENV COMMON specified
		D-series compilers with ENV LIBRARY specified
	FORTRAN	D-series compilers with ENV COMMON specified
	Pascal	D-series compilers by default
		D-series compilers with ENV COMMON specified
		D series compilers with ENV FULL specified
	TAL	D-series compilers with ENV COMMON specified

Table 2-2. Binder Grouping of ENV Directive Parameters (page 2 of 2)

Binder Group	Language	Generated by:
NEUTRAL	C	D-series compilers with ENV EMBEDDED specified
		D-series compilers with ENV LIBSPACE specified
	Pascal	C-series compilers with ENV EMBEDDED specified
		C-series compilers with ENV LIBSPACE specified
		D-series compilers with ENV EMBEDDED specified
		D-series compilers with ENV LIBSPACE specified
	TAL	C-series compilers by default
		D-series compilers by default
		D-series compilers with ENV NEUTRAL specified

Here are the rules for binding modules together:

- You can bind object files that are in the same Binder group; the target object file runs in the same environment as the input object files.
- You can bind object files that include routines from both the OLD and NEUTRAL Binder groups; the target object file runs in a language-specific run-time environment.
- You can bind object files that include routines from both the COMMON and NEUTRAL Binder groups; the target object file runs in the CRE.
- You cannot bind object files that include routines from both the COMMON and OLD Binder groups.

When you bind object files compiled for different environments, each procedure retains its original ENV attribute.

Note. If you specify the ENV NEUTRAL directive in a TAL source file, BINSERV does not allow the resulting object file to be combined (for example, through a SEARCH directive) with object files compiled for the COMMON or OLD Binder groups. This rule prevents files from gaining the COMMON or OLD attributes.

Table 2-3. Run-Time Environment Resulting From Binding Modules

	Binder Group		
Binder Group	**OLD**	**COMMON**	**NEUTRAL**
OLD	language-specific	Not allowed	language-specific
COMMON	Not allowed	CRE	CRE
NEUTRAL	language-specific	CRE	language-specific or CRE

Use the Binder INFO command with the DETAIL clause to show the ENV attribute of a particular data or code block. For example:

```
@ add * from test
@ info *, detail
TEST^PROCEDURE 167
LANG: TAL ENV: NEUTRAL TIME: 1994-06-19 09:48
```

Examples

The following examples show the different combinations for binding C, COBOL85, and TAL modules. Pascal modules can be substituted for C modules and FORTRAN modules can be substituted for COBOL85 modules throughout these examples.

- Binding a C-series C module with a D-series C module

  A C-series C module is a member of the OLD Binder group. A D-Series C module is a member of the COMMON Binder group.

  You cannot bind these two modules together without recompilation. You must recompile the C-series module using a D-series compiler to bind these modules.

- Binding a C-series COBOL85 module with a D-series COBOL85 module

  A C-series COBOL85 module is a member of the OLD Binder group. A D-series COBOL85 module can be a member of either the OLD or COMMON Binder groups.

  You can bind the C-series module with a D-series module compiled without an ENV directive or with an ENV OLD directive.

  Otherwise, you must recompile, recompile the C-series module using a D-series compiler with an ENV COMMON directive.

- Binding a D-series C module with a D-series COBOL85 module

  A D-series C module is a member of the COMMON Binder group. A D-series COBOL85 module can be a member of either the OLD or COMMON Binder groups.

  You must compile the COBOL85 module with an ENV COMMON directive to bind these two modules together.

- Binding a C-series C module with a D-series COBOL85 module

  A C-series C module is a member of the OLD Binder group. A D-series COBOL85 module can be a member of either the OLD or COMMON Binder groups.

  You can bind the C-series C module with a D-series COBOL85 module compiled without an ENV directive or with an ENV OLD directive.

  Otherwise, you must recompile the C-series C module using a D-series C compiler.

- Binding a D-series C module with a C-series COBOL85 module

  A D-series C module is a member of the COMMON Binder group. A C-series COBOL85 module is a member of the OLD Binder group.

  You cannot bind these two modules together without recompilation. You must recompile the C-series COBOL85 module using a D-series compiler with an ENV COMMON directive to bind these modules.

- Binding a C-series TAL module with a D-series TAL module

  A C-series TAL module is a member of the NEUTRAL Binder group. A D-series TAL module can be a member of the OLD, COMMON, or NEUTRAL Binder groups.

  You can bind the C-series module with a D-series module compiled without an ENV directive or with an OLD, COMMON, or NEUTRAL directive.

- Binding a D-series C module with a D-series TAL module

  A D-series C module is a member of the COMMON Binder group. A D-series TAL module can be a member of the OLD, COMMON, or NEUTRAL Binder groups.

  You can bind the D-series C module with a D-series TAL module compiled without an ENV directive or with an ENV COMMON or ENV NEUTRAL directive.

  Otherwise, you must recompile the TAL module without an ENV directive or with an ENV COMMON or ENV NEUTRAL directive.

- Binding a D-series COBOL85 module with a D-series TAL module

  A D-series COBOL85 module can be a member of either the OLD or COMMON Binder groups. A D-series TAL module can be a member of the OLD, COMMON, or NEUTRAL Binder groups.

  You can bind the D-series COBOL85 module with the D-series TAL module if both modules were compiled without an ENV directive or with the same ENV directive. You can also bind a D-series TAL module compiled without an ENV directive or with an ENV NEUTRAL directive with any COBOL85 module.

Binding COBOL85 and FORTRAN Programs

When you create an object file using BIND commands, Binder retains the program unit concepts of the COBOL85 and FORTRAN compilers. Each COBOL85 code block in

the include code block list is a separate program unit. All FORTRAN code blocks that are bound together form one program unit. Based on these concepts, Binder constructs all the control blocks needed to execute the program. These control blocks include:

- One run-unit control block (RUCB) for the target file.
- One program unit control block (PUCB) for each COBOL85 program unit in the target file; the PUCB includes the file control blocks.
- One program unit control block for the FORTRAN program unit in the object file (the FORTRAN special data block, #PUCB, is always the last PUCB data block in any object file).
- One file control block for each file (logical unit) in the set of files for the FORTRAN program unit in the object file; these control blocks are contained in the PUCB blocks.
- One FORTRAN logical unit table.

Binding C Programs

The C compiler generates one bindable code file for each source language compilation unit. C code files are executable only when you specify a RUNNABLE pragma to instruct the compiler to build an executable object file. Generally, you use the RUNNABLE pragma in C to build an executable object file automatically. For large programs created from multiple modules, it is often easier to invoke BIND as a separate step.

To produce an executable C object file without using the RUNNABLE pragma, you must invoke Binder and use it to do the following:

1. Bind together your separately compiled modules.
2. Bind in the appropriate version of the C memory-model file.
3. Bind in the appropriate version of the C run-time library file for C-series programs only.
4. Specify a heap size before issuing the BUILD command. You can specify the heap size in PAGES (1024 words), WORDS, or BYTES.

On C-series systems the C run-time library resides in a file, CLIB, that must be bound into each object file that contains C functions. On D-series systems the C run-time library is split between two files, CLIB and CRELIB. Because CLIB and CRELIB are configured into the system library, do not bind them into object files. The D20 and D30 C run-time libraries have significant architectural differences. Because of these differences, D20 programs that bound in CLIB without also binding in CRELIB will not run correctly on D30 or later product versions. You must bind such programs again without CLIB or CRELIB. CLIB no longer comes as a separate file. Check your BIND scripts to ensure that you do not bind in CLIB or CRELIB. Delete any old copies of CLIB from $SYSTEM.SYSTEM.

To bind a C program, either include the commands in a command file or enter them interactively. The following example shows a sample Binder session for binding a D-series C program:

```
SELECT CHECK PARAMETER LENIENT
ADD * FROM mainobj
ADD * FROM obj1
SELECT SEARCH $system.system.cwide
SELECT RUNNABLE OBJECT ON
SELECT LIST * OFF
SET HEAP 10 PAGES
BUILD exobj
```

Note the following points about this example:

- The file MAINOBJ is the user's main object file.
- The file OBJ1 is a user file.
- The value of 10 PAGES in the SET HEAP command refers to the maximum heap size for the target object file.
- You must set MODE to NOUPSHIFT if you specify code block names and entry point names. MODE NOUPSHIFT directs Binder to differentiate between uppercase and lowercase characters in names.
- This example program uses the wide data model. The memory-model file named after the SELECT SEARCH command would have been named CLARGE if the program used the large memory model or CSMALL if the program used the small memory model. See the *HP C/C++ Programmer's Guide* for additional information.

Binding Pascal Programs

The Pascal compiler generates one bindable code file for each source language compilation unit. Pascal code files are not executable. To produce an executable Pascal object file, you must invoke Binder and use it to do the following:

1. Bind together your separately compiled modules.
2. Bind in the appropriate version of the Pascal run-time library file for C-series programs only.
3. Specify a heap size before issuing the BUILD command. You can specify the heap size in PAGES (1024 words), WORDS, or BYTES.

As with a C program, you can bind a Pascal program by including the commands in a command file or entering them interactively. The following example shows a sample Binder session for binding a D-series Pascal program:

```
ADD * FROM mainobj
SELECT SEARCH (extraobj, moreobj)
SELECT RUNNABLE OBJECT ON
SELECT LIST * OFF
SET HEAP 2 PAGES
BUILD exobj
```

The following example shows a sample Binder session for binding a C-series Pascal program:

```
ADD * FROM mainobj
SELECT SEARCH (extraobj, moreobj)
SELECT SEARCH $SYSTEM.PASCAL.PASRUN
SELECT RUNNABLE OBJECT ON
SELECT LIST * OFF
SET HEAP 2 PAGES
BUILD exobj
```

Note the following points about these examples:

- The file MAINOBJ is the user's main object file.
- The files EXTRAOBJ and MOREOBJ are user files.
- For C-series Pascal programs, you must also specify the correct Pascal run-time library, PASRUN or PASRUNS. Use PASRUN for programs that run in user code space; use PASRUNS for programs that run in system or user library space or that contain a main program written in another language. On C-series systems, PASRUN and PASRUNS reside in the subvolume containing the Pascal compiler. On D-series systems, the Pascal run-time library is named PASLIB. PASLIB resides in the system library. See the *Pascal Reference Manual* for additional information.

Binding Mixed-Language Programs

The rules that govern binding mixed-language programs are determined by the run-time environment. Before the D-series Release Version Updates (RVUs), C, COBOL85, FORTRAN, Pascal, and TAL each had its own run-time environment. These language-specific run-time environments were different from one another and often incompatible. The Common Run-Time Environment (CRE) provides a shared run-time facility for these languages. Under CRE, you can write mixed-language programs without the restrictions imposed by the C-series compilers and run-time environments.

All the object code in a single object file must be either compiled to operate under CRE or compiled to operate under the pre-CRE conventions. Refer to the discussion of Binding Rules on page 2-6 for more information.

There are additional restrictions in the areas of naming global variables and routines, sharing global data, interlanguage procedure calls, and input and output. For more information on these subjects, refer to the *Common Run-Time Environment Programmer's Guide* and the language reference manuals and programmer's guides for the languages you are using.

The following restrictions apply to binding mixed-language programs compiled with C-series compilers or D-series compilers (with the ENV OLD directive):

- You cannot bind programs containing both C and FORTRAN routines.

- If your program contains a COBOL85 module, that module must be the main module.

Parameter Checking for a Mixed-Language Bind

When binding modules from more than one language, you can check for parameter mismatches with the CHECK PARAMETER option of the SELECT command. By default, this option is set to STRICT. When this option is set to STRICT, Binder checks each parameter for proper size, type, and mode (passed by value or reference); it also checks the return type of functions. Binder issues an error message if calls and entry points do not match.

For mixed-language programs, specify SELECT CHECK PARAMETER STRONG to avoid extraneous Binder messages. Refer to the SELECT Command on page 3-49 for more information.

Binding SQL Program Files

When binding SQL program files, you can bind them before running them through either the NonStop SQL/MP compiler or the Accelerator and NonStop SQL/MP compiler. You do so just as you would for any standard language file. For example:

```
@BIND
@ADD * FROM sqlfile
@ADD * FROM mainfile
@BUILD newfile!
```

The difficulty with SQL program files occurs when you attempt to replace SQL procedures in a bound program with new SQL procedures. Each code block in a procedure references corresponding data blocks and run-time data units (RTDUs). Depending on the choice and order of Binder commands, the newly bound file may or may not have the new RTDUs and data blocks.

To ensure that the final program contains the new code blocks, data blocks, and RTDUs for the SQL procedures, use one of the following three methods:

1. Rebuild the entire file using a series of ADD * FROM commands or a single ADD * FROM MAIN command followed by a series of SELECT SEARCH commands for each additional module.

2. Add the new SQL procedures first followed by everything from the old file. For example:

   ```
   @ADD * FROM sqlfile
   @ADD CODE * FROM oldfile
   @BUILD newfile!
   ```

   In this example Binder adds the code blocks, data blocks, and RTDUs to the include list from SQLFILE. When Binder encounters the old SQL entry points in OLDFILE, it issues a warning that the entry points are already on the include list

and does not include the second (and old) copy of the entry points, code blocks, or RTDUs.

Note. Specifying ADD * FROM OLDFILE as the second command would require overhead for the same reason as the command sequence in method #3. See method #3 for a detailed explanation.

The following example also works by specifying the file containing the new procedures first:

```
@ADD CODE * FROM sqlfile
@ADD CODE * FROM oldfile
@BUILD newfile!
```

In this example Binder places the code blocks from SQLFILE on the include list and the corresponding data blocks and RTDUs on the unresolved reference lists. It also notes that the data blocks and RTDUs can be resolved from SQLFILE. After the BUILD command, Binder then resolves the references to the data blocks and RTDUs from SQLFILE.

Note. The previous example does not work if you transpose the two ADD commands. If the ADD CODE * FROM OLDFILE command occurred first, Binder would place the referenced RTDUs on the unresolved reference list with pointers to OLDFILE.

---

3. Add everything from the old file, then replace the SQL code blocks and everything associated with them. For example:

```
@ADD * FROM oldfile
@REPLACE * FROM sqlfile
@BUILD newfile!
```

The REPLACE * command places the code blocks, data blocks, and RTDUs from SQLFILE onto the include lists.

This method causes potential memory problems. Binder names data blocks using the timestamp from the compilation. Therefore, the data blocks in OLDFILE and SQLFILE have different names. The REPLACE command cannot replace the data blocks added with the first command because they have different names from the data blocks referenced in the REPLACE command. REPLACE cannot replace something that is not there. Because the new data blocks are referenced by the new procedure code, Binder puts the new data blocks on the unresolved reference list and resolves the reference to the data blocks from SQLFILE after a BUILD command. NEWFILE then contains both the old data blocks and the new data blocks. If you use this method, you will eventually run out of data space.

In the previous example, a REPLACE CODE command would not work. REPLACE CODE would add the new code blocks to the include list and place references to the data blocks and RTDUs on the unresolved reference lists. After a BUILD command, Binder would then resolve the references to the data blocks, giving you both the old and new data blocks, but it would resolve the references to the RTDUs from OLDFILE. Therefore, NEWFILE would not include the correct RTDUs.

To decrease the time required to SQL compile a program, Binder estimates the size of the SQL object run-time data unit (RTDU) by adding slack space to the object file during the host-language compilation. If the SQL-compiled object fits in the pre-allocated slack space, then Binder can quickly update the RTDU region with the compiled SQL code. Binder estimates the size of the SQL object RTDU region by multiplying by four the size of the SQL source RTDU region. To determine the amount of slack space Binder adds to the object file during host compilation, enter the following Binder commands to list all blocks:

```
@ADD * FROM object-file
@INFO INCLUDE *
```

Next add the size of all blocks of type SQL_SRC, and multiply the value by four.

Resolving External References

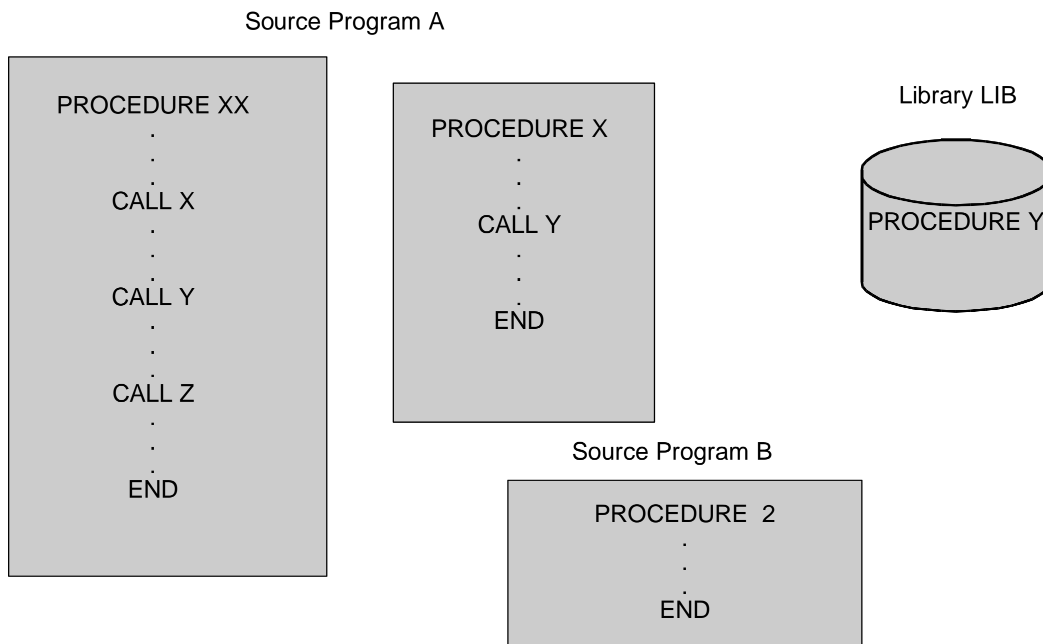
Procedures in source code programs often contain references to other procedures known as external references.

A major part of building an executable object file is resolving external references. Binder resolves external references by locating entry points in other object files, if they exist, and including a copy of any compiled code and data required by the reference in the target file.

References to procedures in other programs, references to procedures in the same source file, and references to a library of procedures shared by many users are all considered external references. If a reference is made to a procedure entry point in the same source file, or to a library of procedures shared by many users, the referenced entry point name is considered external to the procedure. For information about user libraries, see Section 6, User Libraries

If Binder cannot find the entry point to an external reference, the reference is not satisfied. In the example shown in Figure 2-1, Program B can still be in the planning stages when Program A is compiled. In this case, Binder would not be able to resolve the reference to Procedure Z, and it would leave the entry point reference for later resolution. Resolution can occur later in another binding operation or at program load time.

Figure 2-1. External References



Note. External references to system procedures are unresolved until you first execute the program that contains the reference. The user program does not define the external references to the system procedures. When the program is first run, the operating system resolves any unresolved references that are to system procedures. For more information, see Unresolved Reference Lists on page 5-6.

Specifying a Different Volume for Binder Work Files

Binder uses generous amounts of disk space. If there is insufficient disk space for temporary files and the target file, Binder terminates with an error message and does not build a target file. In this case, you want Binder to create its work files on a volume other than the current default volume.

The rules for specifying a volume for Binder work files depend on whether you are using Binder interactively or as part of a compilation.

When Using BIND

When disk space is insufficient, Binder terminates with an error message and does not retain any of the information supplied prior to the BUILD command. You must start over.

To have work files created on a different volume, use the command PARAM SWAPVOL at the TACL prompt. The syntax of the PARAM SWAPVOL command is:

```
PARAM SWAPVOL [ node-name. ] volume-name
```

In the following example, the PARAM SWAPVOL command specifies $SCRATCH as the volume for Binder work files:

```
14> PARAM SWAPVOL $SCRATCH
```

For better performance, keep the swap volume, the BIND process, and the input and output object files on the same node.

Note that the PARAM SWAPVOL command determines the swap volume used by BIND regardless of the language used to code the input files. Thus, if issued, PARAM SWAPVOL controls the swap volume used by BIND when it is processing files coded in any valid language.

When Using BINSERV

If Binder is operating as part of a compilation, the swap volume is determined by whether you entered the TACL command PARAM SWAPVOL before the compilation command. The rule is:

- If a PARAM SWAPVOL command was issued, BINSERV uses the swap volume specified in the PARAM SWAPVOL command.
- If a PARAM SWAPVOL command was not issued, BINSERV uses the same volume that contains the new object file resulting from the compilation as the swap volume.

This is true for all languages except Pascal and C.

Specifying the Swap Volume for Pascal and C Programs

For Pascal and C programs, the rules for swap volumes are different. The `SWAP volume` option of the Pascal and C compilation commands determines the swap volume used by the BINSERV process. The command interpreter PARAM SWAPVOL command has no effect on the swap volume used by BINSERV when it is started by the Pascal or C compilers.

If you do not specify the SWAP volume option in a Pascal or C compilation command, the BINSERV process uses the volume on which the compiler code file resides as the

swap volume. Compiler code usually resides on volume $SYSTEM. See Section 7, Guardian File Names and TACL Commands for more information.

Generating Output Listings

You can generate output listings using the DUMP, INFO, LIST, SHOW, or BUILD command.

Table 2-4. Commands that Produce Listings

Command	Listing Produced
BUILD	Displays target file statistics upon successful completion of a BUILD operation.
DUMP	Displays all or part of the contents of a specified code or data block.
INFO	Displays information about code blocks, entry points, data blocks, and RTDUs in the include, unresolved, and undefined reference lists.
LIST, SELECT LIST, or BUILD...LIST	Displays load map and cross-reference data for code blocks, entry points, data blocks, and RTDUs. By default, load maps are alphabetical.
SHOW	Displays current values for the current file and its attributes, SET and SELECT command parameters, and modifications established by the MODIFY command.

To print these listings to a file, use the OUT command on the BIND command line. In the following examples, the first command creates a location load map for OBJFILEC in file LISTFILE; the second example sends the listings specified in the command file CMDFILE to a printer named $S.#LAND.

```
16>BIND /OUT listfile / LIST LOC FROM objfilec
17>BIND /IN cmdfile, OUT $s.#land/
```

Binder sends the output from any of the commands listed in Table 2-2 to the listing file, if one is specified; otherwise, it displays the output at the home terminal. In interactive mode, use the OUT parameter to capture the output of a single command. The following subsections describe each of the listings.

Target File Statistics

After a BUILD command executes, Binder automatically produces statistics for the constructed target file. These statistics are shown in Figure 2-2 and described in Table 2-5.

Figure 2-2. Target File Statistics

```
BINDER - OBJECT FILE BINDER - T9621D30 - (31OCT94) SYSTEM \HOME
Copyright Tandem Computers Incorporated 1982-1994

Object file \HOME.$DATA.SAMPLE.OBJECT
User library file name $DATA.BIND.LIBRARY
TIMESTAMP 1994-06-19 15:17:12

         3 Code pages

         16 Primary data words

         603 Secondary data words

         35 Data pages

         0 Resident code pages

         20 Extended data pages

            Heap location: Extended data space

         10 Heap size in pages

         619 Top of stack location in words

         1 Code segment

         0 Binder warnings

         0 Binder errors

Elapsed time - 00:00:05
```

Table 2-5. Binder Statistics (page 1 of 2)

Statistic	Definition
Object file name	The name of the constructed target object file
User library file name	The name of the object file to be linked to a program file at run time, if one exists
Object file timestamp	The target file timestamp
Code area size	The number of pages required for code area allocation
Primary data	The number of words of primary data space
Secondary data	The number of words of secondary data space
Data area size	The minimum number of pages required for data area allocation
Resident code size	The number of pages allocated for resident code
Extended data area size	The number of pages allocated for an extended data segment

Table 2-5. Binder Statistics (page 2 of 2)

Statistic	Definition
Heap location	The type of memory that the heap area is located in. This entry appears only for Pascal or C files that have been bound with the Pascal or C run-time libraries, respectively. The heap location can be extended data space or regular data space
Heap size	Heap size in pages
Top of stack	Location of the top of the stack in words
Number of code segments	The number of code segments in the object file
Number of Binder warnings	The number of warning messages issued
Number of Binder errors	The number of error messages issued
Elapsed time	The amount of time (hh:mm:ss) spent in building the object file

Load Maps

Binder can produce separate load maps for data blocks and entry points. By default, load maps are alphabetical. You can also produce location load maps for entry points and data blocks. You use the BUILD LIST, SELECT LIST, or LIST commands to generate these listings.

Entry Point Maps

Entry point maps are load maps that contain storage addresses of code blocks and entry points of a program.

A code block contains the executable machine code for a routine that is invoked through a procedure call (PCAL) or an external call (XCAL) instruction and the procedure entry point (PEP) table.

An entry point is the address or the label of the first instruction that the program or process executes on entering a program, routine, or subroutine. A program or a code block can have many different entry points, each corresponding to a different function or purpose.

Alphabetic Entry Point Load Maps.

Figure 2-3 shows an alphabetically sequenced load map for code blocks and entry points. The Binder wraps lines if the output line length of your listing is greater than the record length of the file.

Figure 2-3. Alphabetic Entry Point Map for Entry Points and Code Blocks

```
ENTRY POINT MAP BY NAME FOR FILE: \KITTY.$BIND.BINDER.TEST

SP PEP BASE   LIMIT  ENTRY ATTRS NAME              DATE       TIME LANGUAGE SOURCE FILE

00 052 033224 033224 033224 V  IN^ASSIGN^MESSAGE   1994-11-19 01:45 TAL  $BIND.Binder.INITIALS
00 033 002070 003401 003021    IN^INITIALIZE       1994-11-19 01:45 TAL  $BIND.Binder.INITIALS
00 051 032775 033223 032775 V  IN^PARAM^MESSAGE    1994-11-19 01:45 TAL  $BIND.Binder.INITIALS
00 050 032651 032774 032651 V  IN^STARTUP^MESSAGE  1994-11-19 01:45 TAL  $BIND.Binder.INITIALS
00 066 035546 035640 035546    IOP^POP^INPUT^STACK 1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 056 033741 034001 033741 V  IO^READ             1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 034 003402 005651 005040    IO^READ^LINE        1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 027 001435 001527 001435    IO^WRITE            1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 063 035236 035255 035236    IO^WRITE^BOTH       1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 030 001530 001552 001530    IO^WRITE^INPUT      1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 042 032050 032075 032050    IO^WRITE^LOG        1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 043 032076 032121 032076    IO^WRITE^SPOOLER    1994-11-19 01:45 TAL  $INSP.PDTSHARE.PINOUT
00 071 050232 050664 050232    LC^ADD              1994-10-24 08:00 TAL  $BIND.LIBRTY.LISTCOMS
00 072 050665 051624 051156    LC^ALTER            1994-10-24 08:00 TAL  $BIND.LIBRTY.LISTCOMS
00 074 052411 052450 052411    LC^BEGIN^TEMPORARY  1994-10-24 08:00 TAL  $BIND.LIBRTY.LISTCOMS
00 073 051625 052410 052046    LC^BUILD            1994-10-24 08:00 TAL  $BIND.LIBRTY.LISTCOMS
00 075 052451 053134 052451    LC^CHANGE           1994-10-24 08:00 TAL  $BIND.LIBRTY.LISTCOMS
00 055 033640 033740 033640 V  LC^CLEAR            1994-10-24 08:00 TAL  $BIND.LIBRTY.LISTCOMS
```

Table 2-6. Information Included in Load Maps for Entry Points (page 1 of 2)

Label	Description
SP	Code segment containing the entry point
PEP	PEP table number
BASE	Base address of the code block defining the entry point
LIMIT	End address of the code block defining the entry point
ENTRY	Address of the entry point
ATTRS	Attributes of the entry point: C = Callable P = Privileged E = Secondary entry R = Resident I = Interrupt V = Variable arguments M = Main X = Extensible arguments The absence of an attribute indicates a primary entry point.
NAME	Entry point name

Table 2-6. Information Included in Load Maps for Entry Points (page 2 of 2)

Label	Description
DATE/TIME	Timestamp for compilation of the block
LANGUAGE	Source language of the block: TAL, FORTRAN, COBOL85, C, or Pascal
SOURCE FILE	Disk file name of source code for the block

Location Entry Point Load Maps

If you specify a command to display entry point maps by location, Binder outputs an itemized listing following each map. The listing contains:

- Code size in words
- Procedure entry point (PEP) length in words
- Global P-relative length in words
- Procedure length in words
- Gap length at 32K boundary in words
- External entry point (XEP) size
- Code area size in pages
- Resident code size in pages

Figure 2-4 shows the location load map for an object file with multiple code segments.

Figure 2-4. Entry Point Map by Location for Multiple Code Segments

```
TIMESTAMP 1994-03-04 15:18:34

ENTRY POINT MAP BY LOCATION FOR FILE: \NORTH.$USERS.QA.SAMPLE
CODE SEGMENT 00

SP PEP BASE   LIMIT  ENTRY  ATTRS NAME      DATE       TIME  LANGUAGE SOURCE FILE

00 002 000006 000065 000006       SUB1      1994-10-24 11:51 FORTRAN  $BUG.QATACL.FORTSUBS

CODE SEGMENT 01

SP PEP BASE   LIMIT  ENTRY ATTRS NAME       DATE        TIME  LANGUAGE SOURCE FILE

01 002 000006 000067 000006      SUB2       1994-10-24  11:51 FORTRAN  $BUG.QATACL.FORTSUBS

CODE SEGMENT 2

SP PEP BASE   LIMIT  ENTRY ATTRS NAME       DATE        TIME  LANGUAGE SOURCE FILE

02 002 000006 000100 000006      SUB3       1994-10-24  11:51 FORTRAN  $BUG.QATACL.FORTSUBS

CODE SEGMENT 03

SP PEP BASE   LIMIT  ENTRY ATTRS NAME        DATE        TIME LANGUAGE SOURCE FILE

03 002 000006 000117 000006      CHARASSIGN^ 1994-10-24  11:51 TAL      $EM1.T9262C20.FTNLIB2
```

Data Block Maps

Figure 2-5 shows an alphabetically sequenced load map for data blocks. At the completion of a BUILD command, Binder puts read-only data blocks in a separate map. If you use a LIST command, Binder lists read-only data blocks with the other data blocks.

Figure 2-5. Alphabetical Load Maps for Data Blocks

```
DATA BLOCK MAP BY NAME FOR FILE: \HOME.$BIND.LIBRTY.SAMPLE

BASE   LIMIT  TYPE   MODE NAME             DATE       TIME  LANGUAGE SOURCE FILE

025117 025166 COMMON WORD .APPLY^          1994-02-14 06:44 TAL      $BIND.LIBRTY.APPLYS
001334 001462 COMMON WORD .APPLY^PUBLIC   1994-02-14 06:44 TAL      $BIND.Binder.APPLYS
042612 045671 COMMON WORD .BINDLST         1994-03-12 04:34 TAL      $BIND.Binder.BINDLSTS
001463 001542 COMMON WORD .BINDLST^PUBLIC  1994-03-12 04:34 TAL      $BIND.Binder.BINDLSTS
025167 042216 COMMON WORD .BLDOBJ          1994-06-19 12:00 TAL      $BIND.LIBRTY.BLDOBJS
001543 004342 COMMON WORD .BLDOBJ^PUBLIC   1994-06-19 12:00 TAL      $BIND.Binder.BLDOBJS
004343 004425 COMMON WORD .BLDTAB^PUBLIC   1994-06-19 12:00 TAL      $BIND.Binder.BLDTABS
004530 005543 COMMON WORD .CC^LOG^DATA     1994-03-12 04:34 TAL      $ADA1.PDTSHARE.PCOMMAND
023577 024576 COMMON WORD .CC^SAVE^DATA    1994-03-12 04:34 TAL      $ADA1.PDTSHARE.PCOMMAND
000366 000510 COMMON WORD .GLOBAL^VAR      1994-06-19 12:00 TAL      $BIND.Binder.GLOBALS
005544 023514 COMMON WORD .IO^DATA         1994-06-19 15:04 TAL      $ADA1.PDTSHARE.PINOUT
042503 042513 COMMON WORD .LISTCOM         1994-02-14 08:44 TAL      $BIND.LIBRTY.LISTCOMS
042514 042611 COMMON WORD .LISTMGR         1994-11-19 19:36 TAL      $BIND.Binder.LISTMGRS
```

Table 2-7. Information Included in Load Maps for Data Blocks

Label	Description
BASE	Base address of the data block (a word address, even for extended storage)
LIMIT	End address of the data block (a word address, even for extended storage; if blank, the block is empty)
TYPE	Type of the data block (own, common, special)
MODE	Word or string
NAME	Name of the data block
DATE	Date of the compilation
TIME	Timestamp for the compilation
LANGUAGE	Source language of the block: C, COBOL85, FORTRAN, Pascal, or TAL
SOURCE FILE	File name of the source code for the block

Figure 2-6 shows the listing for read-only data blocks by name and by location. For additional information about read-only data blocks, see Section 4, Object File Structure

Figure 2-6. Listing for Read-Only Data Blocks

```
READ-ONLY DATA BLOCK MAP BY NAME FOR FILE: \HOME.$KEVIN.LIBRTY.FILE1

SP BASE    LIMIT  TYPE   MODE   NAME                 DATE       TIME  LANGUAGE SOURCE FILE

00 C100024 100062 COMMON WORD   FINDKEY^OFFSET^TABLE 1994-11-19 18:04 TAL     $BIND.Binder.LISTM
00 C123127 123320 COMMON WORD   HASH^DESC            1994-06-19 12:00 TAL     $BIND.LIBRTY.WORDS
00 C125351 125407 COMMON WORD   HASH^TABLE^LENGTH    1994-11-19 18:04 TAL     $BIND.Binder.LISTM
00 C123516 124115 COMMON STRING RWTEXT               1994-06-19 12:00 TAL     $BIND.LIBRTY.WORDS
00 C127765 130023 COMMON WORD   SECONDARY^LIST       1994-11-19 18:04 TAL     $BIND.Binder.LISTM
01 C053152 053210 COMMON WORD   SECONDARY^LIST       1994-11-19 18:04 TAL     $BIND.Binder.LISTM
00 C077761 100017 COMMON WORD   SIZE^NODE^TABLE      1994-11-19 18:04 TAL     $BIND.Binder.LISTM
00 C123321 123515 COMMON WORD   WORD^DESC            1994-06-19 12:00 TAL     $BIND.LIBRTY.WORDS
00 C122732 123126 COMMON WORD   WORD^INFO            1994-06-19 12:00 TAL     $BIND.LIBRTY.WORDS

READ-ONLY DATA BLOCK MAP BY LOCATION FOR FILE: \HOME.$KEVIN.LIBRTY.FILE1

CODE SEGMENT 00

SP BASE    LIMIT  TYPE   MODE   NAME                 DATE       TIME LANGUAGE SOURCE FILE

00 C077761 100017 COMMON WORD   SIZE^NODE^TABLE      1994-11-19 18:04 TAL    $BIND.Binder.LISTM
00 C100024 100062 COMMON WORD   FINDKEY^OFFSET^TABLE 1994-11-19 18:04 TAL    $BIND.Binder.LISTM
00 C122732 123126 COMMON WORD   WORD^INFO            1994-06-19 12:00 TAL    $BIND.LIBRTY.WORDS
00 C123127 123320 COMMON WORD   HASH^DESC            1994-06-19 12:00 TAL    $BIND.LIBRTY.WORDS
00 C123321 123515 COMMON WORD   WORD^DESC            1994-06-19 12:00 TAL    $BIND.LIBRTY.WORDS
00 C123516 124115 COMMON STRING RWTEXT               1994-06-19 12:00 TAL    $BIND.LIBRTY.WORDS
00 C125351 125407 COMMON WORD   HASH^TABLE^LENGTH    1994-11-19 18:04 TAL    $BIND.Binder.LISTM
```

Cross-Reference Lists

The cross-reference lists are produced if you specify one of the following commands:

- SELECT LIST XREF ON or BUILD LIST XREF ON produces a cross-reference listing of code blocks, entry points, and data blocks.
- LIST XREF CODE [name-list] produces a cross-reference listing of code
- blocks.
- LIST XREF DATA [name-list] produces a cross-reference listing of data blocks.
- LIST XREF produces a cross-reference listing of code blocks, entry points, and data blocks.

You can also enter SELECT LIST * ON, BUILD LIST * ON, or LIST *. See Section 3, BIND Commands for details.

Code block listings consist of these items:

- Names and locations of procedures called by the code block
- Names and locations of procedure that the code block calls

Entry point listings consist of these items:

- Entry point name
- Name of the code block containing the reference

- Location of each reference

Data block cross-reference listings consist of these items:

- Data block name (either a common block name or the TAL special block, #GLOBAL)
- Location and storage type (word, byte) of referenced identifier
- Name of the code block containing block references
- Location of each reference

Locations are word offsets, in octal, from the base of the block.

Figure 2-7 shows a sample cross-reference list for code blocks, entry points, and data blocks.

Figure 2-7. Cross-Reference Listing (page 1 of 2)

```
BINDER - OBJECT FILE BINDER - T9621D30 - (31OCT94) SYSTEM \BECCA
Copyright Tandem Computers Incorporated 1982-1994
TIMESTAMP 1994-06-19 11:55:33

ENTRY POINT CROSS REFERENCE
(SORTED BY REFERENCED CODE BLOCK)
------------------------------------

REFERENCED ENTRY POINT    REFERENCING CODE BLOCK    WORD OFFSET OF REFERENCES

PROC1                     M                         00002

                          PROC2                     00002

                          PROC3                     00002

PROC2                     M                         00003

                          PROC3                     00003 00016

PROC3                     M                         00006

PROC4                     M                         00007

STOP                      M                         00012

ENTRY POINT CROSS REFERENCE
(SORTED BY REFERENCING CODE BLOCK)
------------------------------------

REFERENCING CODE BLOCK    REFERENCED ENTRY POINT    WORD OFFSET OF REFERENCES

M                         PROC1                     00002

                          PROC2                     00003

                          PROC3                     00006

                          PROC4                     00007

                          STOP                      00012

PROC2                     PROC1                     00002

PROC3                     PROC1                     00002

                          PROC2                     00003 00016
```

Figure 2-7. Cross-Reference Listing (page 2 of 2)

```
COMMON BLOCK CROSS REFERENCE
(SORTED BY REFERENCING CODE BLOCK)
------------------------------------

REFERENCING CODE BLOCK  OFFSET        TYPE  REFERENCED COMMON BLOCK    WORD OFFSET OF REFERENCES

PROC3                   000000 000000 WORD  #GLOBAL                    00005

                        000000 000001 WORD  #GLOBAL                    00007

                        000000 000002 WORD  #GLOBAL                    00011

                        000000 000001 WORD  DATA1                      00013

                        000000 000002 WORD  DATA1                      00015

PROC4                   000000 000001 WORD  #GLOBAL                    00001

COMMON BLOCK CROSS REFERENCE
(SORTED BY REFERENCED COMMON BLOCK)
-------------------------------------

REFERENCED COMMON BLOCK  OFFSET        TYPE  REFERENCING CODE BLOCK    WORD OFFSET OF REFERENCES

#GLOBAL                  000000 000000 WORD  PROC3                     00005

                         000000 000001 WORD  PROC3                     00007

                                             PROC4                     00001

                         000000 000002 WORD  PROC3                     00011

DATA1                    000000 000001 WORD  PROC3                     00013

                         000000 000002 WORD  PROC3                     00015

ENTRY POINT MAP BY NAME

SP PEP BASE   LIMIT  ENTRY  ATTRS NAME  DATE       TIME  LANGUAGE SOURCE FILE

00 006 000051 000063 000051 M     M     1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

00 002 000007 000021 000021 V     PROC1 1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

00 003 000022 000025 000022       PROC2 1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

00 004 000026 000045 000026 V     PROC3 1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

00 005 000046 000050 000046       PROC4 1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

DATA BLOCK MAP BY NAME

BASE   LIMIT  TYPE   MODE NAME     DATE       TIME  LANGUAGE SOURCE FILE

000000 000002 COMMON WORD #GLOBAL  1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

000006 000006 COMMON WORD .#GLOBAL 1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

000007 000152 COMMON WORD .DATA1   1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1

000003 000005 COMMON WORD DATA1    1994-07-27 11:55 TAL      $BIND.MANUAL.STEST1
```

3 BIND Commands

You can use BIND commands to specify input object files, to define and build the target file, and to query the status of options set by the SET and SELECT commands. The following topics contain an overview of commands and syntax conventions.

Topic	Page
Summary of Most Commonly Used Commands	3-2
How to Use BIND Commands Efficiently	3-3
Syntax Conventions for Name Lists as Command Elements	3-4

This section also describes all the commands available in BIND in alphabetical order.

Command	Page
ADD Command	3-6
ALTER Command	3-9
BUILD Command	3-11
CD Command	3-14
CHANGE Command	3-15
CLEAR Command	3-18
COMMENT Command	3-19
DELETE Command	3-19
DUMP Command	3-20
ENV Command	3-22
EXIT Command	3-23
FC Command	3-23
FILE Command	3-24
HELP Command	3-24
INFO Command	3-25
LIST Command	3-28
LMAP Command	3-32
LOG Command	3-32
MODE Command	3-33
MODIFY Command	3-34
MOVE Command	3-36
OBEY Command	3-38
OUT Command	3-39
RENAME Command	3-40
REPLACE Command	3-41
RESELECT Command	3-43

Command	Page
RESET Command	3-44
SATISFY Command	3-47
SELECT Command	3-49
SET Command	3-57
SHOW Command	3-67
STRIP Command	3-75
SYSTEM Command	3-77
VERIFY Command	3-77
VOLUME Command	3-78

For a summary of BIND commands and their syntax, see Section 9, Syntax Summary.

Summary of Most Commonly Used Commands

BIND commands allow a broad range of object file manipulations.

Table 3-1. Commonly Used BIND Commands (page 1 of 2)

Command	Description
ADD	Names the blocks and entry points to include; replaces blocks, entry points, and any referenced RTDUs, deleting any previous occurrence of the named item.
BUILD	Creates the target file.
CLEAR	Deletes input information without building a file.
COMMENT	Enters comments.
DELETE	Removes block names from the include lists.
DUMP	Displays object file contents in ASCII, HEX, ICODE, OCTAL, or DECIMAL.
FILE	Gives a default disk file name for the current file to use for retrieval of code or data blocks.
INFO	Displays information about names on the include, unresolved reference, and undefined lists.
LIST	Selects load maps and object cross-reference listings.
REPLACE	Names replacements for code blocks, data blocks, and any referenced RTDUs already on the include lists.
SELECT	Specifies controls for satisfying references or building the target file; also selects BIND services such as parameter checking or compressing the code area.

Table 3-1. Commonly Used BIND Commands (page 2 of 2)

Command	Description
SET	Sets object file characteristics of the target file.
SHOW	Displays the current file name and values in effect for SELECT and SET; also displays the modify list. Displays contents of an existing file.
STRIP	Deletes Binder, Inspect, and Accelerator information from the object files.

The following restrictions apply when entering commands:

- You can enter multiple commands on the same line by separating them with the semicolon character (;).
- If you enter a COMMENT in a multiple command line, it must be the last command on the line.
- To continue a command to the next line, enter the ampersand (&) as the last nonblank character of the current line. The maximum line length, including continuation characters, is 132 characters. The maximum length of a continued command record is 528 characters.

BIND also provides the following commands that are commonly used in other tools and utilities:

```
ENV          HELP         OUT
EXIT         LOG          SYSTEM
FC           OBEY         VOLUME
```

Binder automatically expands a partial file name if it contains the appropriate file-name part. For details, refer to Section 7, Guardian File Names and TACL Commands.

How to Use BIND Commands Efficiently

Some BIND commands are more efficient than other commands. This subsection discusses these commands and their less efficient counterparts.

Searching for Files

An ADD * FROM file-name command tends to be more efficient than a SELECT SEARCH command for files with a large number of entry points. The use of ADD * over SELECT SEARCH may result in a performance increase.

Replacing Procedures

When replacing a procedure, use the REPLACE command rather than the ADD command to replace a single module in a large file. This is a more efficient procedure than rebuilding the file using ADD commands for each module contained in the file. For

example, if you built LARGFILE from MOD1, MOD2, MOD3, and MOD4, then altered MOD1, enter the following commands to replace MOD1 in LARGFILE:

```
@ADD * FROM LARGFILE
@REPLACE MOD1
@BUILD LARGFILE!
```

Turning off Load Maps

Turning off load maps with the SELECT LIST * OFF command results in a substantial performance increase. This is because half of the time spent generating a target file is actually spent producing load maps. Turn off load maps only when you are certain that the specified command sequence will produce exactly what you intend. If you are interested in viewing the load maps only up to a specific point, you can press the Break key during load map output instead.

Binding without Fixups

You can save additional time by using SELECT FIXUPS OFF on files that must be processed a second time by Binder. SELECT FIXUPS OFF tells Binder not to fix code and data references in the object file. To create an executable file, you must turn SELECT FIXUPS to ON when binding the file the second time.

Syntax Conventions for Name Lists as Command Elements

Many BIND commands allow lists of names for entry points, code blocks, or data blocks to be used as part of the command line syntax.

Table 3-2. Syntax Conventions for Named Lists (page 1 of 2)

Element	Definition
string	A valid language identifier for a code or data block or any number of leading characters of a code or data block identifier or an octal code or data block address
name-list	*string*
	(*string* [, *string*])
block-name	Valid language identifier for a code block or for a data block
block-range	*block-name*
	block-name TO *block-name*
	* TO *block-name*
	block-name TO *

Table 3-2. Syntax Conventions for Named Lists (page 2 of 2)

Element	Definition
block-list	*block-range*
	(*block-range* [, *block-range*]...)
	*
entry-name	A valid language identifier for a primary or secondary entry point name
entry-range	*entry-name*
	entry-name TO *entry-name*
	* TO *entry-name*
	entry-name TO *
entry-list	*entry-range*
	(*entry-range* [, *entry-range*]...)
	*
*	All members of the current include list or input object file
name TO *name*	All members from the first name to the second name of the include list or object
	file
* TO *name*	All members from the start of the include list or object file to the given name
name TO *	All members from the given name to the end of the include list or object file

Name lists can include one or more code or data blocks. A code block is a C function, a COBOL85 program unit, a FORTRAN program or subprogram, a Pascal routine, or a TAL PROC. A data block is defined differently for each language. For information about data blocks and data block names, see Section 4, Object File Structure.

Because name lists can also include ranges of names within lists, you can specify ranges of blocks or entry points; for example, the ADD, DELETE, and REPLACE commands allow this usage.

Name ranges can apply to input object files. If so, the range is the span of blocks or entry points between the first name and the second name. You must give the names in the order of their location in the file.

You can also specify ranges of names, such as on include lists; for example, the ALTER, INFO, and MOVE commands allow this usage. The range is determined by the order in which the names appear on the include list. You can use the INFO command to display these lists.

ADD Command

The ADD command has two forms. The first form of the ADD command inserts names in the four include lists: include entry point list, include data block list, include code block list, and include run-time data unit list (RTDU list). The second form of the ADD command specifies that a code segment boundary marker be placed at the end of the current include code list.

ADD resolves all previously unresolved external references that are satisfied by the added entry point or data block, except those named on the omit list.

You cannot use the ADD command to include nested code blocks (blocks with lexical levels greater than one).

```
ADD { CODE entry-list } [ FROM file-name ] [ , DELETE ]
{ DATA block-list }
{ * }
or
ADD SPACE
```

`CODE` *`entry-list`*

specifies code to be included in the target file. Associated code blocks, entry points, own blocks, and RTDUs are added to the appropriate include lists for each entry point named. See Table 3-2 for valid forms of entry-list.

`DATA` *`block-list`*

specifies FORTRAN COMMON blocks, TAL BLOCKs, C external variables, Pascal external variables, Pascal exported variables, and COBOL85 EXTERNAL data items to be included in the target file. See Table 3-2 for valid forms of block-list.

`*`

specifies all code blocks, entry points, data blocks, and referenced RTDUs in the file be added to the applicable include lists.

`FROM` *`file-name`*

specifies the Guardian object file name or OSS pathname to use. The default is the current file.

`DELETE`

specifies any previously inserted occurrences of names added by this command be deleted.

`SPACE`

specifies a code segment boundary marker be placed at the end of the current include code list. The `ADD SPACE` command cannot include any other parameters.

Usually, Binder minimizes the number of TNS code segments used, by filling each 64K-instruction segment as fully as possible. The rarely-used `ADD SPACE` command causes Binder to leave the current code segment unfilled, sending all procs from subsequent include lists or search lists into the next segment(s). Consecutive `ADD SPACE` commands will create one or more empty segments. For example, when SYSGEN invokes BIND to create the TNS SC System Code library, SYSGEN uses `ADD SPACE` commands to have the first nonempty segment be addressed as SC.06 instead of the default SC.00. Another possible use for the `ADD SPACE` command or the `MOVE IN NEW SPACE` command is to manually arrange that frequently-called procs are located in the same segment as their primary callers. A program codefile cannot have more than 32 TNS code segments, numbered 0..31.

Considerations

- Binding Modules Compiled for Different Run-Time Environments

  For a description of the rules Binder applies when binding modules compiled for different run-time environments, see Section 2, Using Binder

- Entry Point or Data Block Names Not on the Include Lists

  If you specify an entry or data block name for a block not already on an include list, the ADD command adds the name to the end of the list, regardless of whether you specify DELETE. If you specify an entry or data block name for a block already on the include list, ADD adds the name to the end of the list only if you also specify the DELETE option. Otherwise, Binder ignores the ADD command. For entry points, Binder also issues a warning message indicating that the specified entry point is already on the include list.

- Entry Point in an ADD Command

  Naming an entry point in an ADD command automatically inserts the code block that contains the entry point in the include code block list. Usually, data blocks are added implicitly (by means of references in included code). You can add FORTRAN COMMON blocks, TAL BLOCKs, C external variables, Pascal public variables, and COBOL85 EXTERNAL blocks explicitly. This option lets you define a different set of initial values.

- Pascal Procedures

  If you want to use the ADD, DELETE option of the ADD command to replace one copy of a Pascal procedure with another copy that has the same name, you must export the procedure in both modules. Otherwise, Binder considers them to be separate procedures. For example, if module A and module B both contain procedure C and you do not export the procedure in each module, Binder names one A.C and the other B.C. In this case, the command ADD CODE C FROM B, DELETE adds B.C to the file without deleting A.C.

- Target File Order

To ensure Binder builds the target file correctly, add names to the lists in the order in which you want them to appear in the target file. Binder adds names to the include lists in the order in which ADD commands specify them and, in general, uses the include list order to build the target file.

- COBOL85 Files

  For COBOL85 files, ADD CODE is equivalent to ADD *, because COBOL85 files do not have separate data blocks except for COBOL85 EXTERNAL.

- Creating an Object File that Has Multiple Code Segments

  ADD SPACE lets you create an object file that has multiple code segments by adding code blocks from one object file, adding a space boundary marker, adding more blocks from the same file or another file, inserting another space, and so on.

- Rules for Setting HIGHPIN, HIGHREQUESTERS, and RUNNAMED

  Binder sets HIGHPIN ON for a target object file only if all of the files that make up the target object file are set HIGHPIN ON. Binder sets HIGHREQUESTERS ON for a target object file if and only if HIGHREQUESTERS ON is set for the object file containing the main program. Binder sets RUNNAMED ON for a target object file if any object file that makes up the target file is set RUNNAMED ON.

- Error Conditions

  If a specified entry point or data block is not in the specified FROM file, Binder issues an error message and halts execution of the ADD command.

- Similarity Between the REPLACE Command and the ADD,DELETE Command

  ADD,DELETE adds a specified entry name to the end of the include list, deleting the previous occurrence of the entry; REPLACE removes the previous occurrence of the entry name and inserts the new reference in its place. If an error occurs when you attempt an ADD command, you might accomplish your goal by using a REPLACE command. (Use REPLACE with caution for COBOL85 and FORTRAN files. For details, see the REPLACE Command on page 3-41.)

Examples

- In this example, the first ADD command transfers the contents of OLDFILE to NEWFILE; the second ADD command includes BLK-1 to BLK-5 of OBJ in NEWFILE.

```
@FILE oldfile

@ADD *

@ADD CODE blk-1 TO blk-5 FROM obj

@BUILD newfile
```

- In the following example, NEWFILE contains BLOCK-1 from OBJFILE and all of OLDFILE except any code block named BLOCK-1.

```
@FILE oldfile
@ADD * FROM oldfile
@ADD CODE block-1 FROM objfile, DELETE
@BUILD newfile
```

- The following example shows the use of the ADD SPACE command to create a file containing multiple code segments: code blocks from OLDFILE are in code segment 0, those from SOMEFILE are in code segment 1, and those from ANYFILE are in code segment 2.

```
@FILE oldfile
@ADD * FROM oldfile
@ADD SPACE
@ADD * FROM somefile
@ADD SPACE
@ADD * FROM anyfile
@BUILD newfile
```

ALTER Command

The ALTER command changes the attributes of code blocks and entry points in both the include code block and the include entry point lists. Attributes that can be changed are CALLABLE, MAIN, PRIVILEGED, and RESIDENT. You can change the MAIN attribute only for TAL procedures and C functions.

You cannot use the ALTER command with nested code blocks (blocks with a lexical level greater than one).

```
ALTER entry-list , alter-spec [ , alter-spec ] ...
alter-spec is one of:
CALLABLE { ON | OFF }
LIKE entry-name
MAIN { ON | OFF }
PRIV { ON | OFF }
RESIDENT { ON | OFF }
```

entry-list

specifies one or more primary or secondary entry points on the corresponding include lists. See Table 3-2 for valid forms of *entry-list*.

alter-spec

is one of the following code attributes:

`CALLABLE {ON | OFF}`

specifies whether privileged entry points are callable by nonprivileged procedures. CALLABLE applies only to privileged code, and Binder automatically sets PRIV ON for an entry point with CALLABLE ON.

`LIKE` *`entry-name`*

specifies that the entry points in entry-list have the same attributes as LIKE entry-name, which must be on an include list. If LIKE occurs, it overrides any preceding parameters of this ALTER command.

`MAIN {ON | OFF}`

specifies whether the entry points in entry-list have the MAIN attribute. You can specify MAIN only for TAL and C code; COBOL85, Pascal, and FORTRAN MAIN characteristics are set permanently at compilation time.

`PRIV {ON | OFF}`

specifies whether entry points in entry-list are run in privileged mode. If PRIV ON is set, the procedure can be called only by procedures that also run in privileged mode.

`RESIDENT {ON | OFF}`

specifies whether code blocks reside in main memory the entire time the process is running. Binder automatically applies RESIDENT to the code block containing the named entry point.

Examples

The following examples illustrate the syntax of the ALTER command.

- In the following example, the attributes of the SUB^1 code block are changed to match the attributes of SUB^2.

```
@ADD CODE sub^1 FROM objfile1

@ADD CODE sub^2 FROM objfile2

@ALTER sub^1, LIKE sub^2
```

- The ALTER command in this example changes the RESIDENT attribute to ON for all code blocks and entry points from the beginning of the include lists through entry name SUB^5.

```
@ALTER * TO sub^5, RESIDENT ON
```

- The next example changes the CALLABLE attribute to OFF for entry names SUB^1 and SUB^5 through SUB^8 on the include lists.

```
@ALTER (sub^1, sub^5 TO sub^8), CALLABLE OFF
```

BUILD Command

The BUILD command builds the target file using the code block, entry point, data block, and RTDU names from the include lists. You enter the BUILD command after completely defining the target file with other commands. After the build operation, Binder returns to its initial state.

```
BUILD [ / OUT file-name / ] [ file-name ] [ ! ]
      [ , { set-param | select-param } ] ...
```

OUT *file-name*

directs the output listing to the specified file. See the description of the OUT Command on page 3-39 for additional information.

file-name

is a valid Guardian file name or OSS pathname for the target file. If *file-name* already exists, Binder cannot write to it unless you use the *!* option, which tells Binder to purge the existing file.

In the Guardian environment, Binder uses the default file name OBJECT in the default volume and subvolume if you omit *file-name*. Binder also uses the file name OBJECT if you did not specify *!* and *file-name* either already exists or cannot be accessed.

In the OSS environment, Binder uses the default file name object in the default working directory if you omit *file-name*. Binder also uses the file name object if you do not specify *!* and file-name either already exists or cannot be accessed.

!

specifies that Binder purge any previously existing file named *file-name*.

set-param

specifies an object-file attribute for the target file. This specification overrides, for this command only, any value previously established for that attribute.

set-param can be one of the following:

{DATA} *value* [PAGES | WORDS | BYTES]

{EXTENDSTACK}

{STACK}

HEAP *value* [PAGES | WORDS | BYTES]

HEAP STATISTICS {ON | OFF}

HIGHPIN {ON | OFF}

HIGHREQUESTERS {ON | OFF}

`INSPECT {ON | OFF}`

`LARGESTACK` *`value`* `[PAGES | WORDS | BYTES]`

`LIBRARY` *`file-name`*

`LIKE` *`file-name`*

`PEP` *`value`*

`PFS` *`value`* `[PAGES | WORDS | BYTES]`

`RUNNAMED {ON | OFF}`

`SAVEABEND {ON | OFF}`

`SUBTYPE` *`number`*

`SYMBOLS {ON | OFF}`

`SYSTYPE {GUARDIAN | OSS}`

`TARGET [TNS | TNS/R | TNS/E | ANY]`

`USERLIBRARY {ON | OFF}`

See the SET Command on page 3-57 for a description of these specifiers.

`select-param`

specifies a Binder control or option for building the object file. This specification overrides, for this command only, any value previously established for that parameter. *`select-param`* is defined as one of the following:

`{CHECK` *`check-option`*`}`

`{CHECK (`*`check-option`* `[,` *`check-option`*`]...)}`

`COMPACT {ON | OFF}`

`COMPRESS DATA {ON | OFF}`

`FILESYS {OSS | GUARDIAN}`

`FIXUPS {ON | OFF}`

`{LIST` *`listing-option`*`}`

`{LIST (`*`listing-option`* `[,` *`listing-option`*`]...)}`

`{OMIT` *`entry-name`*`}`

`{OMIT (`*`entry-name`* `[,` *`entry-name`*`]...)}`

`{REFER` *`refer-pair`*`}`

`{REFER (`*`refer-pair`* `[,` *`refer-pair`*`]...)}`

`RUNNABLE OBJECT {ON | OFF}`

```
SATISFY {ON | OFF}
{SEARCH file-name}
{SEARCH (file-name [, file-name]...)}
WARNINGS {ON | OFF}
```

See the SELECT Command on page 3-49 for a description of these specifiers.

Considerations

- Using BUILD with SATISFY ON

  If you enter BUILD SATISFY ON, Binder resolves any remaining external references. Binder accesses the user files on the search list and, if it finds any of the unresolved entry points, adds the corresponding code to the target file it is building.

  If you enter BUILD SATISFY OFF, Binder does not resolve remaining external references, but does resolve data references. The default is BUILD SATISFY ON.

- Target File Name

  Binder creates the target file in a temporary file and gives the target file one of the following names:

  - *file-name*, if a file by that name did not exist.
  - *file-name*, if a file by that name previously existed, but you specified *!*, and the purge was successful.
  - OBJECT, if no *file-name* was specified (default), or if *file-name* existed and was not purged.
  - *ZZBI nnnn*, where *nnnn* is a random numeric identifier, if the preceding naming attempts failed.

  Note. If the old object file is running when you specify a new object file of the same name with the ! option, Binder does not purge the old file but renames it ZZBI*nnnn* (where *nnnn* is supplied by Binder).

- Error Conditions

  The build operation fails if Binder cannot name the target file. When this happens, Binder issues an error message and prompts you for input. The temporary file containing the target file is lost, as is the information from previous commands. You must specify the target file contents again by entering definition commands.

  If the build operation failed because of insufficient disk space, you must correct the condition before the build operation can succeed.

The only error condition that does not cause BUILD to clear the lists (and, consequently, the only one that does not require you to begin again) is ILLEGAL SYNTAX.

Examples

The following examples illustrate the syntax of the BUILD command.

- This example uses the REPLACE command to replace code for the entry point EPNAME1 in the object file FILENAME with code of the same name from NEWFILE.

```
@ADD * FROM filename
@REPLACE CODE epname1 FROM newfile
@BUILD objfile, SEARCH (lib1, lib2)
```

- The following example uses the SATISFY OFF option to suppress resolution of external code references.

```
@ADD CODE sub^1 TO * FROM objfile1
@ADD CODE * FROM objfile2, DELETE
@BUILD newfile, SATISFY OFF
```

CD Command

The CD command allows you to specify the default current working directory Binder uses to expand partial pathnames. The current working directory is only used if the file system is set to OSS in the SELECT FILESYS command. To display the current working directory, use the ENV command.

```
CD { directory }
```

directory

specifies an OSS directory name.

Example

The following example illustrates the syntax of the CD command.

```
CD /usr/smith
```

CHANGE Command

The CHANGE command allows you to amend or patch the attribute values of an existing object file. In this respect, the CHANGE command is similar to the SET command, which specifies attribute values for a target file before it is built.

```
CHANGE
{ AXCEL ENABLE { ON | OFF } } IN file-name
{ DATA value [ PAGES | WORDS | BYTES ] }
{ HIGHPIN { ON | OFF } }
{ HIGHREQUESTERS { ON | OFF } }
{ INSPECT { ON | OFF } }
{ LIBRARY file-name }
{ MISALIGN { FAIL | NOROUND | SYSDEFAULT} IN file-name
{ OCA ENABLE { ON | OFF } } IN file-name
{ PFS value [ PAGES | WORDS | BYTES ] }
{ RUNNAMED { ON | OFF } }
{ SAVEABEND { ON | OFF } }
{ SYSTYPE { GUARDIAN | OSS } }
{ SUBTYPE number }
{ TARGET { TNS | TNS/R | TNS/E | ANY } }
{ USER BUFFER { ON | OFF } }
```

`AXCEL ENABLE {ON | OFF}`

changes the AXCEL ENABLE attribute to ON or OFF. By default, Binder sets the AXCEL ENABLE attribute to OFF. The Accelerator automatically sets the value of this attribute to ON regardless of the Binder setting.

After a file has been processed by the Accelerator, AXCEL ENABLE OFF disables the Accelerator region of an object file, allowing you to execute the TNS program code in accelerated mode for certain debugging purposes. This is usually unnecessary.

`DATA` *`value`* `[PAGES | WORDS | BYTES]`

specifies the amount of data space to be allocated for the object file. The default data space allocated is the maximum number of data pages in any of the files from which data is included or the number of pages needed to hold all the data blocks plus an estimate of the stack space needed for local storage, whichever is larger. If value is less than the value already in the target file, it is ignored.

You can specify either a decimal value or an octal value (preceded by %) for *value*.

You can specify any of the following units for value: PAGES, WORDS, or BYTES. The default unit for *value* is PAGES. (One PAGE is 1024 WORDS.)

If you specify BYTES as the unit and enter an odd number for value, Binder rounds this number up to an even value.

`HIGHPIN {ON | OFF}`

specifies whether an object file can run at a high process identification number (PIN), if the process creation request allows it and a high PIN is available.

`HIGHREQUESTERS {ON | OFF}`

specifies whether an object file can support calls from requesters running at a high process identification number (PIN).

`INSPECT {ON | OFF}`

specifies whether the Inspect program or the Debug program is chosen for debugging when you execute the object file. The default is OFF; that is, the Debug program is used. INSPECT OFF automatically causes Binder to set SAVEABEND OFF. (You can use the TACL SET INSPECT or RUN command to override the INSPECT option.)

`LIBRARY` *`file-name`*

specifies the name of a user library to be associated with the object file at run time. You can override *`file-name`* at run time by using the LIB parameter in the command interpreter RUN command. The default is no user library.

`MISALIGN { FAIL | NOROUND | SYSDEFAULT} IN` *`file-name`*

specifies the `MISALIGN` attribute in the specified file. This codefile attribute is ignored when executing on TNS/R systems. The attribute is effective on TNS/E systems. The default is `SYSDEFAULT`.

`MISALIGN FAIL` causes the system to generate an instruction failure interrupts, which will be reported as trap 1 (INSTRUCTIONFAILURE) or a process abend if no ARMTRAP handler in a Guardian process. `MISALIGN FAIL` is only effective on TNS/E systems.

`MISALIGN NOROUND` conveniently allows all valid TNS programs and also erroneously-coded TNS programs to run, but not necessarily with the same results as on some prior machines. This option causes the system to complete the operation using the operand's "natural"(unrounded) address.

`MISALIGN SYSDEFAULT` allows application of the current TNS Misalign policy.

`OCA ENABLE { ON | OFF } IN` *`file-name`*

The OCA ENABLE attribute has dual purposes:

It suppresses the translation (OCA will refuse to translate a file that has this bit set). OCA ENABLE OFF will not allow an already Itanium-augmented file to be translated again, returning an error. Again, OCA will refuse to translate an object file (previously not Itanium-augmented) whose OCA ENABLE attribute has been changed to OFF through Binder.

The OCA ENABLE attribute controls whether a program will execute in translated mode on TNS/E machines. By default, Binder sets the OCA ENABLE attribute to ON. OCA does not change the value of this attribute. After a file has been processed by the OCA, OCA ENABLE OFF disables the Itanium region of an object file, allowing you to execute the TNS program code (but not the translated Itanium code) on TNS/E machines for debugging purposes.

`PFS` *`value`* `[PAGES | WORDS | BYTES]`

specifies the size of the process file segment field in the object header of the specified file. *`value`* can be from 64 to 512 pages. The default unit for this command is PAGES. Binder rounds bytes up to the nearest word.

`RUNNAMED {ON | OFF}`

specifies whether an object file runs as a named process, even if no RUN command NAME parameter is specified.

`SAVEABEND {ON | OFF}`

specifies whether a save file is to be created if the process terminates abnormally during execution. Binder automatically sets INSPECT ON if SAVEABEND is ON. The default is OFF.

`SUBTYPE` *`number`*

specifies a value for the process subtype associated with the object file; *`number`* is a decimal value in the range 0 through 63.

`SYSTYPE {GUARDIAN | OSS}`

specifies whether the target execution environment for the object file is the Guardian environments or the OSS environment. The default is Guardian.

`TARGET {TNS | TNS/R | TNS/E | ANY}`

specifies the TARGET attribute in the specified file. See the SET Command on page 3-57 for details. The default is unspecified.

`IN` *`file-name`*

specifies the Guardian object file or OSS pathname to be changed.

`USER BUFFER { ON | OFF }`

changes the value of USER BUFFER from ON to OFF (default is OFF) for any particular object file.

Considerations

- You cannot patch the current file in Binder. If you attempt to use CHANGE on the current file, you must then execute a CLEAR command and start again. Otherwise,

the file remains the current file and repeated attempts to CHANGE result in the same error.

- Each successive CHANGE command specifying one of these parameters overrides the previous specification. Use the SHOW command with the SET *attribute* option to determine the current values of the file attributes.
- CHANGE can be used on files without a Binder region.
- The HIGHPIN ON attribute specifies that an object file is allowed to run at a high PIN. It does not specify that an object file meets the conditions to run at a high PIN. See Running Processes at a High PIN on page 3-65 for more information.
- The HIGHREQUESTERS ON attribute specifies that an object file is allowed to support requests from processes running at high PINs. It does not specify that an object file meets the conditions to support requests from processes running at high PINs. See Running Processes at a High PIN on page 3-65 for more information.

Examples

The following examples illustrate the syntax of the CHANGE command.

- The following command specifies that a save file for MYFILE is to be created if the process terminates abnormally during execution:

```
@CHANGE SAVEABEND ON IN myfile
```

- The following command specifies that the user library LIBFILE is to be associated with MYFILE at run-time:

```
@CHANGE LIBRARY libfile IN myfile
```

- The following command shows the sequence of actions you need to perform in order to find out the current value of an attribute, change the value of the attribute, and verify that the attribute has been changed:

```
@file cref.cobext
CURRENT FILE IS $DATA.CREF.COBEXT
@show set subtype
SUBTYPE 0
@clear
@change subtype 1 in cref.cobext
@file cref.cobext
CURRENT FILE IS $DATA.CREF.COBEXT
@show set subtype
SUBTYPE 1
```

CLEAR Command

The CLEAR command returns Binder to the original state without building an object file.

Binder clears its internal lists (include, omit, refer, search, unresolved reference, undefined, and modify), the current file as established by a FILE command, and all SET and SELECT options.

```
CLEAR
```

COMMENT Command

The COMMENT command allows you to enter comment information to be displayed in the output listing.

```
COMMENT [ text ]
```

text

is a string of characters.

Considerations

- If you use the COMMENT command in a line containing other commands, COMMENT must be the last command on the line.
- To continue COMMENT text over more than one line, start each successive line with either the COMMENT command or the ampersand (&) character.

Examples

The following examples show three COMMENT commands.

```
@DELETE CODE block1; COMMENT deletes all of block1
@COMMENT adds subprog1, subprog2, subprog3 from filea
@COMMENT adds subprog4, subprog5 from fileb
```

DELETE Command

The DELETE command removes the specified blocks from the include lists. If any MODIFY commands were previously specified for these blocks, Binder also removes these changes from the modify list.

You cannot use the DELETE command to delete nested code blocks (code blocks with a lexical level greater than one).

```
DELETE { CODE block-list }
{ DATA block-list }
{ * }
```

`CODE block-list`

specifies code blocks to be deleted. See Table 3-2 for valid forms of block-list.

`DATA block-list`

specifies data blocks to be deleted. See Table 3-2 for valid forms of block-list.

`*`

specifies that all blocks are to be deleted.

Considerations

- The DELETE command has three functions:
  - Places external references to deleted blocks on the unresolved reference lists.
  - Removes external references from deleted blocks from the unresolved and undefined reference lists if no other blocks refer to those names.
  - Discontinues parameter checking for the deleted blocks.
- If you specify DELETE *, the SELECT and SET specifications remain in effect. Binder clears the include, modify, undefined, and unresolved reference lists.

Example

The following example deletes all data and code blocks from the include lists.

```
@DELETE *
```

DUMP Command

The DUMP command displays all or part of the contents of a specified code or data block from the current object file. The display is unformatted.

You can use DUMP on nested code blocks (blocks with a lexical level greater than one); however, if you do not qualify the procedure name, BIND selects the first one in the object file.

```
DUMP [ / OUT file-name / ] { CODE code-block-name }
                           { DATA data-block-name }
{ offset [ , count ] } [ spec-list ] [ FROM file-name ]
{ offset [ , *     ] }
{ *                  }
```

`OUT` *`file-name`*

directs the output listing to the specified file. See the OUT Command on page 3-39 for additional information.

`CODE` *`code-block-name`*

specifies a code block in either the FROM *`file-name`* or the current file.

`DATA` *`data-block-name`*

specifies a data block in either the FROM *`file-name`* or the current file.

`offset`

is a word offset, in octal, from the base of the block.

`, count`

specifies the number of words, in octal, to display starting from *`offset`*. The default is one word.

`, *`

specifies that the part of the block starting at offset and continuing to the end of the block be displayed.

`*`

specifies that the entire block be displayed.

`spec-list`

is one or more format specifiers in the form:

`dump-spec`

`(`*`dump-spec`*` [, `*`dump-spec`*`]...)`

`dump-spec`

is one of the following:

`ASCII`

`DECIMAL`

`HEX`

`ICODE`

`OCTAL`

Use ICODE with the CODE *`code-block-name`* parameter only. The default is OCTAL.

`FROM` *`file-name`*

is the disk file name of an object file. The default is the current file. OSS pathnames are accepted.

Considerations

DUMP displays the current file contents. Therefore, you cannot use DUMP to display file contents changed with a MODIFY command. (MODIFY affects only the target file.) DUMP DATA does not give useful information for COBOL85 data blocks, except for COBOL85 EXTERNAL.

Examples

The following examples illustrate the syntax of the DUMP command.

- This example uses the DUMP command to dump all of BLOCK1.

```
@DUMP CODE block1 * HEX
```

- This example uses the DUMP command to dump eight words from decimal word 12 of BLOCK5 in OBJFILE.

```
@DUMP DATA block5 14,10 ASCII FROM objfile
```

- This example shows the corrective action taken when the current file does not correspond to the dump request.

```
@FILE f1; ADD CODE b1 TO b5; FILE f2; ADD *

@DUMP CODE b3 *

***** ERROR ***** Block does not exist in file: B3

@DUMP CODE b3 * FROM f1
```

ENV Command

The ENV command displays the current settings of program environment parameters.

```
ENV [ LOG       ]
    [ MODE      ]
    [ SYSTEM    ]
    [ VOLUME    ]
    [ DIRECTORY ]
```

Considerations

If you specify no options, Binder displays the values for all parameters. For additional information, see the descriptions of the CD Command, LOG Command, MODE Command, SYSTEM Command, and VOLUME Command.

Example

The following example displays the current settings of the program environment parameters.

```
@env
LOG       (Logging turned off)
MODE      UPSHIFT
SYSTEM    \CALIF
VOLUME    $DATA.FTRAN
DIRECTORY /G/DATA/FTRAN
```

EXIT Command

The EXIT command stops the BIND process. If you do not issue a BUILD command before exiting, Binder does not create a target file.

```
EXIT
```

Considerations

Entering CTRL/Y also stops the BIND process.

FC Command

The FC (Fix Command) command allows you to edit or repeat a command. When you enter FC, the last command you typed appears followed by the FC prompt, a period (.), on the next line. At the prompt you can enter the subcommands R, I, and D to replace, insert, and delete characters in the command line. See the Guardian User's Guide for more information.

```
FC
```

Example

This example uses the FC command to correct a word, from LISP to LIST, in the command line.

```
@LISP SOURCE
^
***** ERROR ***** Invalid syntax
@FC
@LISP SOURCE
. T
@LIST SOURCE
```

FILE Command

The FILE command establishes the current file for subsequent ADD, DUMP, LIST, REPLACE, and SHOW commands. The current file remains the current file until another FILE, CLEAR, or BUILD command is successfully executed. There is no default for the current file.

```
FILE file-name
```

file-name

is the Guardian object file name or OSS pathnames.

Example

The following example establishes $PROGS.PRIMES.X as the current file.

```
@FILE $progs.primes.x
```

HELP Command

The HELP command displays Binder commands and syntax.

```
HELP [ / OUT file-name / ] [ topic [ subtopic [ subtopic ] ] ]
                           [ subtopic                       ]
                           [ < param-name >                 ]
```

OUT *file-name*

directs the help output to the specified file. See OUT Command on page 3-39 for additional information.

topic

is a Binder command name or topic.

subtopic

is a command parameter or Binder topic as displayed in a HELP topic command.

<param-name>

is a command parameter as displayed in a HELP topic command.

Considerations

- If you specify no options for the HELP command, Binder displays the names of all commands and topics.
- You can obtain the correct syntax for a particular subtopic by first specifying the topic with which the parameter is associated; for example:

```
@HELP DELETE
DELETE {CODE <block-list>}
{DATA <block-list>}
{*}
```

- A command parameter can be preceded and followed by optional angle brackets. In the following example, the name of the parameter for which you want help information is surrounded by angle brackets:

```
@HELP DELETE
DELETE {CODE <block-list>}
{DATA <block-list>}
@HELP <block-list>
```

- If you do not get HELP information, verify that the PDTHELP file resides in the same volume and subvolume as Binder. If the file is missing, Binder cannot print HELP information.
- To exit HELP or move up a level within HELP, press RETURN or CRTL/Y.

INFO Command

The INFO command displays information about code blocks, entry points, data blocks, and RTDUs in the include, unresolved, and undefined reference lists.

You can use the INFO command to display information about nested code blocks (blocks with a lexical level greater than one). If you do not specify the qualified block name, however, Binder selects the first occurrence of the block name.

```
INFO [ / OUT file-name / ]
{ INCLUDE { CODE block-list } [ , DETAIL ] }
{         { DATA block-list }               }
{         { ENTRY entry-list}               }
{         { * }                             }
{                                           }
{ UNRESOLVED { DATA }                       }
{            { ENTRY }                      }
{            { * }                          }
{                                           }
{ UNDEFINED *                               }
{                                           }
{ * [ , DETAIL ]                            }
```

OUT *file-name*

directs the output listing to the specified file. See OUT Command on page 3-39 for additional information.

INCLUDE CODE *block-list*

displays the attributes and lengths of code blocks in the block-list and the total code size. See Table 3-2 for valid forms of block-list. With the DETAIL option, only the external definition of a code block is displayed. You can display any instance of such a code block by fully qualifying the block-list with the name of its source file.

INCLUDE DATA *block-list*

displays the lengths of all data blocks in block-list and the total data size.

INCLUDE ENTRY *entry-list*

displays the attributes of entry points in entry-list. See Table 3-2 for valid forms of entry-list.

INCLUDE *

displays the attributes of all code blocks, data blocks, entry points, and RTDUs in the include lists. This is the only way to display all instances of a multiply-defined code block.

DETAIL

provides further information for blocks in the include lists such as date and time compiled, source language and environment (setting of the ENV directive), SYMBOLS ON or OFF, and the source file from which the block was compiled.

UNRESOLVED DATA

displays names of data blocks on the unresolved reference list and in the undefined list.

UNRESOLVED ENTRY

displays the names and default files of entry points in the unresolved reference list.

UNRESOLVED *

displays the names of entry points and data blocks in the unresolved reference lists and the undefined list.

UNDEFINED *

displays the names of data blocks in the undefined data list: data blocks whose initializing module has not been located.

*

displays all include, unresolved, and undefined lists.

Examples

The following examples illustrate the syntax of the INFO command.

- The following example shows the insertion of a code segment boundary in a multiple-code segment:

```
@INFO INCLUDE CODE *

INCLUDE CODE: 18 ENTRIES

NAME                          SIZE    ATTRIB

T9621D30^31OCT94^22JAN95         1

LM^ADDENTRY                    108

         NEW SPACE

LM^ALLOCATE                     69

...

TOTAL CODE SIZE =              986
```

- This example displays all include and unresolved lists including DETAIL information:

```
@INFO *, DETAIL
INCLUDE CODE: 15 ENTRIES
NAME                     SIZE    ATTRIB
FORMATTER                4672
LANG: TAL ENV: NEUTRAL TIME: 1994-02-12 19:19 SYMBOLS: ON
FILE: \USA.$RLSD.TOOLS.VERS3
TOTAL CODE SIZE =        9876
...
INCLUDE ENTRY: 20 ENTRIES
NAME          OFFSET   ATTRIB
FORMATTER       4450   V
LANG: TAL ENV: NEUTRAL TIME: 1994-02-12 19:19 SYMBOLS: ON
FILE: \USA.$RLSD.TOOLS.VERS3
...
INCLUDE DATA: 5 ENTRIES
NAME               SIZE
#RUCB                77
LANG: FORTRAN ENV: OLD TIME: 1992-04-12 19:19 SYMBOLS: ON
FILE: \USA.$RLSD.TOOLS.VERS4
...
TOTAL DATA SIZE =   512
UNRESOLVED ENTRY: 15 ENTRIES
NAME                FILE
UT^INTERNAL^ERROR
...
UNRESOLVED DATA: 0 ENTRIES
```

LIST Command

The LIST command displays load maps and cross-reference data for entry points and code and data blocks. You can specify the object file either with a previous FILE command or with the FROM parameter of the LIST command.

The LIST command prints the object file name and timestamp at the start of its listings.

You can use the LIST command with nested code blocks (blocks with a lexical level greater than one); however, if you do not fully qualify the code block, Binder selects the first occurrence of the code block name in the file.

```
LIST [ / OUT file-name / ]

{ { SOURCE                              } [ FROM file-name     ]      }
{ {                                     }                             }
{ { CODE name-list [ IN SPACE num ]     }                             }
{ {                                     }                             }
{ { CODE block-list                     }                             }
{ {                                     }                             }
{ { DATA name-list                      }                             }
{ {                                     }                             }
{ { DATA block-list                     }                             }
{ {                                     }                             }
{ { XREF [ XREF- options            ] }                               }
{                                                                     }
{ ( list-option [ , list-option ] ... )                               }
{ [ FROM file-name ] [ , BRIEF ]                                      }
```

`OUT` *`file-name`*

directs the output listing to the specified file. *`file-name`* is a standard file name. See OUT Command on page 3-39 for additional information.

`SOURCE`

specifies that the source file be listed for each code and data block used in creating the object file. The output shows, by source file, all code blocks (identified by the letter P) and all data blocks (identified by the letter B) in the object file.

`CODE` *`name-list`* `[IN SPACE` *`num`*`]`

lists the load map for each name in *`name-list`*. The 132-character output gives all the information shown in a load map from Binder. See Table 3-2 for valid forms of *`name-list`*.

`num`

is an octal number between 0 and %37.

`CODE` *`block-list`*

lists the base address in code space for each block name in *`block-list`*. The asterisk specification is not acceptable for this command, but all other valid forms of *`block-list`*, as shown in Table 3-2, are allowed. The 132-character output gives all the information shown in a load map from Binder.

`DATA` *`name-list`*

lists the load map for each address in name-list. The 132-character output gives all the information shown in a load map from Binder. See Table 3-2 for valid forms of *`name-list`*.

`DATA` *`block-list`*

lists the base address in data space for each block name in *`block-list`*. The asterisk specification is not acceptable for this command, but all other valid forms of *`block-list`*, as shown in Table 3-2, are allowed. The 132-character output gives all the information shown in a load map from Binder.

`XREF [`*`XREF-options`*`]`

displays cross-reference information for one or more code blocks, entry points, and data blocks. (The * option in list-options also produces a cross-reference listing.) *`XREF-options`* can be:

`CODE [`*`name-list`*`]`

`DATA [`*`name-list`*`]`

`CODE` *`name-list`*

displays cross-reference information for the specified code blocks. See Table 3-2 for valid forms of *`name-list`*.

`DATA` *`name-list`*

displays cross-reference information for the specified data blocks.

`list-option`

specifies the type of map to be displayed:

`ALPHA [CODE | DATA]`

`LOC [CODE | DATA]`

`*`

`ALPHA [CODE | DATA]`

displays a load map in alphabetic order. The map lists names and addresses for code blocks and data blocks. The map also gives the language and name of the source file that yielded each block and the date and time of compilation.

The CODE option specifies that only the code blocks be printed. The DATA option specifies that only the data blocks be printed.

`LOC [CODE | DATA]`

displays a load map in location order. The map lists names and addresses for code blocks and data blocks. The map also gives source information as for ALPHA. The CODE option specifies that only the code blocks be printed. The DATA option specifies that only the data blocks be printed.

`*`

generates all three listings: ALPHA, LOC, and XREF.

`FROM` *`file-name`*

specifies the Guardian file name or OSS pathname of the object file to be mapped. The default is the current file.

`BRIEF`

requests display of an 80-character load map line rather than the standard 132-character load map line. The truncated line omits the DATE, TIME, LANGUAGE, and SOURCE FILE information for code and data blocks. (Cross-reference information is still displayed with a 132-character format.)

Considerations

If you specify an output device that is a spooler, a line printer, or a disk file, Binder prints the page number, file name, file timestamp, and column header at the start of each page for the LIST SOURCE, LIST ALPHA, LIST LOC, LIST XREF and LIST * commands.

Examples

The following examples illustrate the syntax of the LIST command.

- The following command outputs to LISTFILE a code block, entry point, and data block cross-reference list for the current file.

```
@LIST / OUT listfile / XREF
```

- The following command outputs to the home terminal all three load maps (ALPHA, LOC, XREF) for OLDFILE, using an 80-character truncated line for the alphabetic and location load maps.

```
@LIST * FROM oldfile, BRIEF
```

- The following command outputs to the home terminal the alphabetic and location load maps for OBJFILE, using the standard 132-character line.

```
@LIST (ALPHA, LOC) FROM objfile
```

- The following command outputs to the home terminal the alphabetic load map for OBJFILE, using the standard 132-character line. Only the data blocks are listed.

```
@LIST ALPHA DATA FROM objfile
```

LMAP Command

The LMAP command produces an alphabetical load map of a specified file.

```
LMAP [ / OUT list-file / ] FROM file-name
```

list-file

specifies the name of the file to which Binder sends the output listing.

file-name

specifies the Guardian file name or OSS pathname to be mapped.

Considerations

The LMAP command is useful for debugging stripped files. Unlike the LIST command, the LMAP command does not require the presence of a Binder region.

Example

The following example shows the output produced by the file $BECKY.BINDER.LMAPFILE:

```
@lmap from $becky.binder.lmapfile
TIMESTAMP 1994-02-12 19:19
SP PEP BASE   LIMIT  ENTRY ATTRS NAME
00 005 000143 000147 000143      DIV
00 004 000136 000142 000136      MUL
00 003 000131 000135 000131      SUM
00 002 000006 000130 000006  M   TMAIN
@
```

LOG Command

The LOG command writes a copy of the current session's input and output to a file.

```
LOG { TO file-name }
    { STOP         }
```

TO *file-name*

identifies a file to receive the copy of the commands and output. If the file does not exist, Binder creates a disk file with the name file-name.

`STOP`

closes the current log file and stops all logging.

Considerations

You start logging by entering a LOG command that specifies a file name.

- If *`file-name`* has the form of a disk file name and the file does not exist, Binder creates an EDIT file.
- If the named file is an existing disk file, Binder appends log output to the file.
- If logging is already in progress, Binder closes the previous log file and begins logging to the new file. If *`file-name`* is the same as the previous log file, Binder ignores the LOG command and continues logging to the same file.
- *`file-name`* cannot be an OSS pathname.

MODE Command

The MODE command tells Binder whether it must differentiate between uppercase and lowercase characters in block names and entry point names. You must set MODE to NOUPSHIFT if you specify code blocks or entry points by name for C routines.

```
MODE { UPSHIFT | NOUPSHIFT }
```

`UPSHIFT`

specifies that Binder need not differentiate between uppercase and lowercase characters. This is the default.

`NOUPSHIFT`

specifies that Binder must differentiate between uppercase and lowercase characters, as in C language code.

Considerations

- The ENV command shows the setting of the MODE command.
- Specify a MODE NOUPSHIFT command when binding mixed-language programs that contain C routines.
- If you are in NOUPSHIFT mode, you need to be careful when entering code or data block names from languages other than C in Binder commands. The block names must be in uppercase characters for Binder to be able to find them in the object file.

MODIFY Command

The MODIFY command changes the values of words in the code and data blocks of the target file. No changes are made to existing files.

You can use MODIFY with nested code blocks (blocks with a lexical level greater than one); however, if you do not specify a fully qualified name, Binder selects the first occurrence of the code block name in the file.

```
MODIFY { CODE code-block-name }
{ DATA data-block-name }
[ modify-spec ] [ offset ] [ , value ]...
```

CODE *code-block-name*

specifies the name of a code block in the include code block list to be modified. Use the INFO INCLUDE CODE * command to display the include code block list.

DATA *data-block-name*

is the name of a data block in the include data block list to be modified. Use the INFO INCLUDE DATA * command to display the include data block list.

modify-spec

specifies the input format for value and the output format for the current values. The default is OCTAL. *modify-spec* is one of the following:

ASCII

DECIMAL

HEX

OCTAL

offset

is the offset, in octal, from the base of the block of the word to receive value. The default is the base of the block (offset 0).

value

is an expression that replaces the word contents. Binder prompts (interactively) for the new *value* if you omit it and assumes value is in the same format as *modify-spec*. You must enclose an ASCII value in quotation marks.

If you specify more than one *value*, Binder modifies word locations sequentially.

Considerations

- Prompting Sequence.

The prompting sequence varies as follows:

- If you specify `value` in the MODIFY command, Binder does not prompt you for input. Instead it replaces the current value of the word at the location specified in offset with the new value specified in value.
- If you do not specify `value` but do specify `offset`, Binder prompts for input by displaying the address, in octal, of the location and the current value at that location.
- If you specify neither `offset` nor `value`, Binder prompts for input by displaying the address of the base of the block (offset 0) and the current value at that location.

  The display that you see in prompt mode is as shown below, where *nnnnn* is the offset, and the arrow is a special prompt. You should either enter a replacement value after the arrow or press the carriage return key to indicate that you do not want to modify the block.

  {CODE} *block-name+ nnnnn (old-value)* <--

  {DATA}

  You can select whether the value display is octal, decimal, hex, or ASCII. Binder accepts your replacement value in the same form as the displayed value. Do not specify a prefix for hex or octal values; Binder does not accept numbers in the form % *nnn*.

  Prompting continues until you enter a carriage return (to indicate no further modifications) or until the end of the block occurs.

- Modifying External References

  Binder issues a warning message if you modify a CALL or a reference to global data. If you include the modified target file in a subsequent binding operation, Binder reissues the warning message.

- Verifying Changes

  To verify changes, use the SHOW MODIFY command. Binder changes only the target file, not the input object file.

Examples

The following examples illustrate the syntax and use of the MODIFY command.

- The MODIFY command causes the prompting sequence to begin with the base address. (The resulting display follows the command.)

```
@MODIFY CODE block-3
CODE BLOCK-3+00000 (012345) <-- 012346
CODE BLOCK-3+00001 (000000) <-- CR
@
```

  Note that in the first line of the display BLOCK-3 is the name of the code block, +00000 is the offset of the word, (012345) is the current value, and 012346 is the replacement value entered by the user.

  Note also that in the second line of the display the user enters a carriage return after the arrow and thus stops the prompting for replacement values.

- The SHOW MODIFY command allows the user to verify changes by displaying all modified entries.

```
@MODIFY CODE lm^init
CODE LM^INIT+00000 (070402) <-- 70401
CODE LM^INIT+00001 (024700) <-- 24701
CODE LM^INIT+00002 (002005) <-- 2006
CODE LM^INIT+00003 (040001) <-- 40000
CODE LM^INIT+00004 (014404) <-- 14403
CODE LM^INIT+00005 (100777) <-- CR
@SHOW MODIFY
MODIFY 5 ENTRIES:
MODIFY CODE LM^INIT+00000 = 070401 LADR L+001
MODIFY CODE LM^INIT+00001 = 024701 PUSH 701
MODIFY CODE LM^INIT+00002 = 002006 ADDS +006
MODIFY CODE LM^INIT+00003 = 040000 LOAD G+000
MODIFY CODE LM^INIT+00004 = 014403 BAZ +003
```

MOVE Command

The MOVE command relocates code blocks within the target file. For example, if Binder was unable to fill the gap preceding the 32K boundary because of the order of the include lists or because of code block sizes, you can establish a more efficient order.

You can also use the MOVE command to specify multiple code segments within the target file.

You can use the MOVE command to reduce page faults by grouping procedures that your program frequently uses. Before you do this, you should analyze your program behavior carefully. For more information on obtaining the analytical data needed to analyze program behavior, see the *Measure Reference Manual*.

You cannot use the MOVE command with nested code blocks (blocks with a lexical level greater than one).

```
MOVE entry-list { AFTER entry-name  }
                { BEFORE entry-name }
                { IN NEW SPACE      }
     [ , entry-list { AFTER entry-name  } ]...
                    { BEFORE entry-name }
                    { IN NEW SPACE      }
```

`entry-list`

is a list of one or more entry names in the include entry name list; Binder accepts either a primary or secondary entry point name. If you specify a secondary entry point name, Binder moves the containing block for the secondary entry point name.

You cannot use the asterisk (*) specification, but any other list format shown in Table 3-2 is acceptable.

`AFTER entry-name`

specifies the position in the include entry name list after which entry-list is to appear; entry-name cannot be within the range of entry-list. Binder accepts primary and secondary entry point names.

`BEFORE entry-name`

specifies the position in the include entry name list before which entry-list is to appear; entry-name cannot be within the range of entry-list. Binder accepts primary and secondary entry point names.

`IN NEW SPACE`

tells Binder to end the current code segment without filling it, and to place the procedures of entry-list into a new code segment. See also the related ADD Command `(ADD SPACE)` command. A program codefile cannot have more than 32 TNS code segments, numbered 0 to 31.

Examples

The following examples illustrate the syntax of the MOVE command.

- MOVE repositions the entry name BLOCK-1 from whatever position it currently occupies in the include list to the position immediately following BLOCK-5.

```
@ADD * FROM objfile
@MOVE block-1 AFTER block-5
@BUILD
```

- MOVE repositions the entry names BLOCK-1 and BLOCK-7 from whatever positions they currently occupy in the include list to the positions immediately preceding BLOCK-2.

```
@MOVE (block-1, block-7) BEFORE block-2
```

OBEY Command

The OBEY command tells Binder to read commands from a specified file and directs output listing to a specified file.

```
OBEY [ / OUT file-name / ] file-name
```

`OUT` *`file-name`*

directs any output listing to the specified file. Error messages are also displayed at the terminal if you are using Binder interactively, however. For additional information, see OUT Command on page 3-39.

`file-name`

is the name of the OBEY file.

Considerations

- Binder reads commands from the named file and processes them until it encounters an end of file. At end of file, Binder closes the OBEY file, and command input reverts to the previous input file, normally the home terminal.
- If a command in the OBEY file contains an OUT specification, it overrides the OUT specification in the OBEY command. Similarly, any subsequent command in the file that has an OUT specification overrides a previous one.
- Additional OBEY commands can appear within an OBEY file; you can nest OBEY files up to a depth of four.
- If an OBEY file changes the default setting of the node or volume, that setting remains in effect for all subsequent commands, including commands entered after you return from the OBEY file. To return to the previous settings, you must enter another SYSTEM or VOLUME command.

- If any part of the specification is invalid, if the file does not exist, or if the file cannot be opened, Binder displays an error message and prompts for input if the input file is a terminal. If the input file is not a terminal, Binder terminates.
- If Binder detects an error while processing an OBEY file, it closes the file and, in the case of nested OBEY files, any other OBEY files currently open. If the original input file was a terminal, Binder issues a prompt on the terminal. If the input file was not a terminal, Binder terminates.

OUT Command

The OUT command directs the output listing to a specified file.

```
{ OUT file-name                             }
{ command / OUT file-name / param-name }
```

file-name

is a standard Guardian file name.

command

is a Binder command.

param-name

specifies one or more parameters for command.

Considerations

- The first form of the OUT command causes permanent redirection of the output. To redirect the output back to the terminal, enter another OUT command with the terminal name as the *file-name*.
- The second form of the OUT command causes temporary redirection of the output. You use this form as part of another command.
- If the file name has the form of a disk filename and the file does not exist, an EDIT file is created. If the named file is an existing disk file, the output is appended to the file.
- If you specify an invalid file name or if Binder cannot open the file, Binder displays an error message and does not execute the command.
- *file-name* cannot be an OSS pathname.

Examples

The following examples illustrate the syntax of the OUT command.

- The following example directs the output produced by the SHOW command to the printer $S.#LP1, then redirects it back to the terminal named $TERM.

```
@OUT $S.#LP1
@SHOW INFO FROM myfile
@OUT $TERM
```

- The following example directs the HELP description of the BUILD command to the printer $S.#LP3. Any output subsequent to this command automatically goes to the previous output setting.

```
@HELP BUILD /OUT $s.#lp3 /
```

RENAME Command

The RENAME command renames a code or data block.

You cannot use RENAME with nested code blocks (code blocks with a lexical level greater than one).

```
RENAME { CODE entry-point-name } TO name
       { DATA data-block-name  }
```

`CODE` *`entry-point-name`*

specifies the current name of the code block to rename.

`DATA` *`data-block-name`*

specifies the current name of a data block to rename.

`name`

specifies the new name assigned to the specified block.

Considerations

- Use the RENAME command when binding together two object files that contain different types of data blocks with the same name. For example, one data block is an OWN block, and the other is a COMMON block.
- You are not allowed to rename special data blocks.

Example

The following example renames BLOCK1 to BLOCKA.

```
@RENAME CODE block1 TO blocka
```

REPLACE Command

The REPLACE command inserts replacements for code blocks, data blocks, and any referenced entry points or RTDUs into the include lists.

You cannot use the REPLACE command with nested blocks (blocks with a lexical level greater than one).

```
REPLACE { CODE entry-list } [ FROM file-name ]
        { DATA block-list }
        { *               }
```

CODE *entry-list*

specifies entry points in the file to replace entry points in the include entry name list. See Table 3-2 for valid forms of entry-list.

DATA *block-list*

specifies data blocks in the file to replace data blocks in the include data block list. See Table 3-2 for valid forms of block-list.

*

specifies that all entry points, data blocks, and RTDUs in the file are to replace matching entry points, data blocks, and RTDUs in the appropriate include lists.

FROM *file-name*

specifies the Guardian file name or OSS pathname containing the entry points and data blocks to replace corresponding ones in the include lists. The default is the current file.

Considerations

- Binder sets HIGHPIN ON for a target object file if and only if all of the files that make up the target object file are set HIGHPIN ON.
- Binder sets HIGHREQUESTERS ON for a target object file if and only if HIGHREQUESTERS ON is set for the object file containing the main program.
- Binder sets RUNNAMED ON for a target object file if any object file that makes up the target file is set RUNNAMED ON.
- The ADD command (with the DELETE option specified) performs functions similar to those performed by REPLACE. ADD,DELETE adds a specified entry name to the end of the include list, deleting the previous occurrence of the entry. REPLACE removes the previous occurrence of the entry name and inserts the new reference in its place.
- If an error occurs during a REPLACE command, you might accomplish the operation by using ADD,DELETE. For example, the REPLACE command replaces

both the direct and the indirect data blocks in a TAL object file. In this case, however, you can successfully use the ADD command with the DELETE option.

- If you want to replace one copy of a Pascal procedure with another of the same name, you must export the procedure in each module containing it. Otherwise, Binder leaves the old copy of the procedure in the file.
- The REPLACE CODE command replaces the specified code blocks and puts any references to data blocks and entry points on the unresolved list. The REPLACE * command replaces all code blocks, data blocks, and entry points.

  Before replacing any language or SQL code blocks with the REPLACE CODE command, you need to understand completely the interdependencies between an object file and Binder. The choice and order of commands affects whether Binder resolves references to the data blocks and entry points from the file containing the new code blocks or from the file containing the old code blocks.

  For more information on how Binder resolves unresolved references, see Unresolved Reference Lists on page 5-6.

Examples

The following examples illustrate the syntax of the REPLACE command.

- In the following example, the REPLACE command replaces the code designated by the entry name BLOCK-1 in OLDFILE with code by the same entry name in OBJFILE. Note that you must enter an ADD command at some point in your session before you enter a REPLACE command. Otherwise, the include lists do not contain any blocks that can be replaced by the blocks specified in the REPLACE command.

```
@ADD * FROM oldfile
@REPLACE CODE block-1 FROM objfile
```

- Assume that the object file OBJFILE was compiled with a SEARCH directive for library LIBFILE. Later library LIBFILE was recompiled, and so OBJFILE must be rebuilt. In the following example OBJFILE is rebuilt by replacing the old contents of LIBFILE with the new contents of LIBFILE, and the rebuilt file is given the name NEWOFILE.

```
@ADD * FROM objfile
@REPLACE * FROM libfile
@BUILD newofile
```

- Code or data specified in the REPLACE command but not on include lists is ignored and no warning message is issued. For example, you entered the following commands:

```
@ADD * from x
@REPLACE * from y
```

If procedure SUBPROG1 is in y, but not in x, then SUBPROG1 is not added.

- The following example replaces a FORTRAN subprogram in OBJFILEA with a new subprogram in OBJFILEB. In this case, a command file contains the Binder commands.

```
13> FORTRAN / IN newsub2, OUT listfile / objfileb
14> BIND /IN cmdfile, OUT listfile /
15> RUN trgfilec
```

CMDFILE contains the following Binder commands:

```
COMMENT - all entries from objfilea are needed
ADD * FROM objfilea
COMMENT - sub2 replaced by new sub2 in objfileb
REPLACE CODE sub2 FROM objfileb
BUILD trgfilec
```

RESELECT Command

The RESELECT command resets one or more SELECT command parameters to the default value. SELECT parameters specify Binder operation during the execution of BUILD and SATISFY commands.

```
RESELECT { select-param [ , select-param ] ... }
         { *                                   }
```

select-param

is one of the following SELECT parameter names:

```
CHECK
REFER
COMPACT
RUNNABLE OBJECT
COMPRESS DATA
SATISFY
FILESYS
FIXUPS
SEARCH
LIST
WARNINGS
```

OMIT

*

specifies all SELECT parameters are to be reset to the default values.

Considerations

The SELECT command parameter defaults are:

CHECK	BLOCK ON, LIBRARY OFF, PARAMETER STRICT
COMPACT	ON
COMPRESS DATA	ON (C, Pascal, COBOL85)
	OFF (TAL, FORTRAN)
FILESYS	GUARDIAN
FIXUPS	ON
LIST	ALPHA ON, LOC OFF, XREF OFF
OMIT	Empty list
REFER	Empty list
RUNNABLE OBJECT	OFF
SATISFY	ON
SEARCH	Empty list
WARNINGS	ON

Examples

- The following command sets all SELECT command parameters to their default values:

```
@RESELECT *
```

- The following RESELECT command clears omit and refer lists, and resets the LIST parameter to its default value:

```
@RESELECT OMIT, REFER, LIST
```

RESET Command

The RESET command resets one or more target file attributes that were previously specified with the SET command to their default values.

```
RESET { set-param [ , set-param ] ... }
      { *                             }
```

set-param

is one of the following SET command parameters:

```
DATA
EXTENDSTACK
HEAP
HEAP STATISTICS
HIGHPIN
HIGHREQUESTERS
INSPECT
LARGESTACK
LIBRARY
LIKE
PEP
PFS
RUNNAMED
SAVEABEND
STACK
SUBTYPE
SYMBOLS
SYSTYPE
TARGET
USERLIBRARY
USER BUFFER
```

*

specifies that all SET attributes be reset to their default values.

Considerations

- Resetting LIKE causes Binder to reset the four parameters DATA, INSPECT, LIBRARY, and SAVEABEND to their default values.
- Resetting the HEAP parameter clears any HEAP size specified. If you do not specify a HEAP size, the HEAP data block is the size of the largest HEAP data block encountered by Binder.

- The SET command parameter default values are:

DATAPAGES	Estimate for all data blocks and stack
EXTENDSTACK	PAGES estimated
HEAP	(No default)
HEAP STATISTICS	OFF
HIGHPIN	OFF
HIGHREQUESTERS	OFF
INSPECT	OFF
LARGESTACK	(No default)
LIBRARY	No user library
LIKE	(No default)
PEP	Minimum for entry points in target file
PFS	128
RUNNAMED	OFF
SAVEABEND	OFF
STACK	PAGES estimated
SUBTYPE	No process subtype
SYMBOLS	ON
SYSTYPE	GUARDIAN
TARGET	Unspecified
USERLIBRARY	OFF
USER BUFFER	OFF

Examples

The following examples illustrate the syntax of the RESET command.

- You have used the SET LIBRARY command to specify user library LIBFILE is to be associated with the object file at run time. Also that you have used the SET SAVEABEND ON command to specify a save file is to be created if the process terminates abnormally during execution. The following command resets the LIBRARY parameter to its default value of no user library and also resets the SAVEABEND parameter to its default value of OFF.

```
@RESET LIBRARY, SAVEABEND
```

- The second example resets all the SET command parameters to their default values.

```
@RESET *
```

SATISFY Command

The SATISFY command tries to resolve external references to entry points and data blocks in the unresolved reference list. Binder uses the current include lists and the search list.

```
SATISFY { select-param                                 }
        { ( select-param [ , select-param ] ... ) }
```

select-param

specifies a SELECT command parameter and value to be used for this statement only. This specification overrides any value previously established for that parameter. select-param is defined as one of the following:

{CHECK *check-option*}

{CHECK (*check-option* [, *check-option*]...)}

COMPACT {ON | OFF}

COMPRESS DATA {ON | OFF}

FILESYS {GUARDIAN | OSS}

FIXUPS {ON | OFF}

{LIST *listing-option*}

{LIST (*listing-option* [, *listing-option*]...)}

{OMIT *entry-name*}

{OMIT (*entry-name* [, *entry-name*]...)}

{REFER *refer-pair*}

{REFER (*refer-pair* [, *refer-pair*]...)}

RUNNABLE OBJECT {ON | OFF}

SATISFY {ON | OFF}

{SEARCH *file-name*}

{SEARCH (*file-name* [, *file-name*]...)}

WARNINGS {ON | OFF}

See the SELECT Command on page 3-49 for a description of these specifiers.

Enter HELP SELECT for a description of these specifiers.

Considerations

- Specifying HIGHPIN ON with the SATISFY Command

Binder sets HIGHPIN ON for a target object file only if all of the files that make up the target object file are set HIGHPIN ON.

- Specifying HIGHREQUESTERS ON with the SATISFY Command

  Binder sets HIGHREQUESTERS ON for a target object file only if HIGHREQUESTERS ON is set for the object file containing the main program.

- Specifying RUNNAMED ON with the SATISFY Command

  Binder sets RUNNAMED ON for a target object file if any object file that makes up the target file is set RUNNAMED ON.

- Using the SEARCH Parameter with the SATISFY Command

  During execution of a SATISFY command, Binder searches the object files listed in the search list in an attempt to resolve any unresolved external references listed in the unresolved reference list. You can add object file names to the search list with the SELECT command, but if you specify the SEARCH *file-name* parameter for the SATISFY command, Binder searches the files you specified instead, overriding any previously established search list.

  If Binder cannot resolve a reference, the reference remains on the unresolved reference list until you enter another SATISFY command or BUILD command that resolves it.

- Using Search Files from the ADD Command

  Binder also uses object files specified in ADD commands as search files for unresolved references, and the order in which Binder searches files might be different from what the user expects.

  See Unresolved Reference Lists on page 5-6 for details.

- Using SATISFY in Interactive Mode

  In interactive mode, Binder prompts for additional target file definition commands following execution of a SATISFY command. Target file generation does not begin until you enter a BUILD command.

  Any *select-param* used in a SATISFY command temporarily sets a new value that Binder uses only during execution of that same SATISFY command.

Examples

The following examples illustrate the syntax of the SATISFY command.

- The following example causes all references to EPNAME1 in OLDFILE to refer instead to EPNAME2 from NEWFILE.

```
@COMMENT - "caller" inserted on next command
@COMMENT - calls epname1; the satisfy will
@COMMENT - result in calling epname2
@ADD CODE caller FROM objfile
@SATISFY SEARCH newfile, REFER epname1 TO epname2
@ADD CODE epname1 FROM oldfile
@BUILD newfile !
```

- The following example leaves entry references unresolved because SATISFY OFF (a select-param) is valid.

```
@ADD * FROM objfile
@SELECT SATISFY OFF
```

- In the following example, Binder searches the files OBJECTC and OBJECTD for external reference resolution, and not files OBJECTA and OBJECTB, which have already been added to the search list with the SELECT command.

```
@SELECT SEARCH (objecta, objectb)
@SATISFY SEARCH (objectc, objectd)
```

- In the following example, if FOO exists in FILE 1 and FILE2, the following commands cause the reference to FOO to be resolved from FILE1 instead of FILE2.

```
@ADD * from file1
@SATISFY SEARCH (file2, file1, file3)
```

SELECT Command

The SELECT command sets parameter values that control Binder during execution of subsequent BUILD and SATISFY commands. You can override these values using the BUILD and SATISFY commands or reset them to default values using the RESELECT command.

```
SELECT { select-param [ , select-param ]... }
```

select-param

is any of the parameter values set with SELECT. You can set one or more parameter values with a single SELECT command. The *select-param* can be any one of the following:

{CHECK *check-option*}

```
{CHECK (check-option [, check-option]...)}
COMPACT {ON | OFF}
COMPRESS DATA {ON | OFF}
FILESYS {OSS | GUARDIAN}
FIXUPS {ON | OFF}
IMPORT LIBRARY library-file-name
{LIST listing-option}
{LIST (listing-option [, listing-option]...)}
{OMIT entry-name}
{OMIT (entry-name [, entry-name]...)}
{REFER refer-pair}
{REFER (refer-pair [, refer-pair]...)}
RUNNABLE OBJECT {ON | OFF}
SATISFY {ON | OFF}
{SEARCH file-name}
{SEARCH (file-name [, file-name]...)}
WARNINGS {ON | OFF}
```

CHECK *check-option*

specifies the type of error checking as one of the following:

```
{BLOCK {ON | OFF}}
{DUPLICATE {ON | OFF}}
{LIBRARY {ON | OFF}}
{PARAMETER {ON | STRICT | STRONG | LENIENT | OFF}}
{WIDEMEM {ON | OFF}}
{*}
```

BLOCK {ON | OFF}

specifies whether Binder checks common blocks for consistency in length and addressing (that is, all byte or all word).

Use CHECK BLOCK ON to check FORTRAN procedures for adherence to the FORTRAN rules for common blocks. CHECK BLOCK ON checks TAL blocks (if a global data block used in several object files changes but only

some of the object files have the new one, Binder detects the length mismatch and issues a warning). Binder also issues a warning if the modules' descriptions of a data block are not consistent. The default is ON.

`DUPLICATE {ON | OFF}`

specifies whether Binder reports a warning if a C data block is declared multiple times. When combining independently developed subsystems, this can be a symptom of accidentally using the same global variable name for two different purposes.

`LIBRARY {ON | OFF}`

controls whether Binder enforces rules to which user libraries must adhere. Use this option when building a target file that will be used as a user library. Binder issues a warning for any reference to a data block other than a read-only block or any entry point that has the MAIN attribute. BIND does not check existing libraries, only user libraries that are being built. The default is OFF.

`PARAMETER {ON | STRICT | STRONG | LENIENT | OFF}`

specifies the extent to which Binder checks parameter lists, function return values, and language consistency between called and calling routines. Binder issues a warning if calls and entry points are not consistent. The default setting is STRICT.

Binder provides five levels of parameter and return-value checking:

ON	Formal and actual parameters must be the same size, type, and mode (value or reference). Return values are not checked.
STRICT	STRICT is the same as ON.
STRONG	Interlanguage calls are checked as LENIENT. Calls between routines compiled from the same language are checked as STRICT.
LENIENT	Parameters and return values are viewed as members of classes. Corresponding formal and actual parameters and corresponding return values must belong to the same class.
OFF	No parameter type and return-value checking.

Binder checks for language consistency between called and calling procedures for all levels of parameter checking except CHECK PARAMETER OFF. Binder checks to make sure that the caller specifies the correct language for the called routine. If the caller explicitly states that the language of the invoked routine is unspecified, Binder does not perform this check.

Refer to Using CHECK PARAMETER on page 3-55 for more information.

`WIDEMEM {ON | OFF}`

specifies whether Binder checks wide and non-wide memory attribute conflicts in the constituent object files when it builds the target object file. The default is OFF.

`* {ON | OFF}`

specifies whether all check options are selected.

`COMPACT {ON | OFF}`

specifies whether Binder fills the gap at the 32K boundary when it builds the target file. A gap occurs when Binder repositions a code block that would cross the 32K boundary to begin at the 32K boundary. The default is ON.

`COMPRESS DATA {ON | OFF}`

specifies whether Binder compresses the regular data space in the object file to include only those pages with initialization data. Binder always compresses the extended data space. The default is ON for C, Pascal, and COBOL85; it is OFF for TAL and FORTRAN; it is always OFF for shared run-time libraries.

`FILESYS {OSS | GUARDIAN}`

specifies whether the Guardian default volume and subvolume or the OSS default directory is used to resolve partial file names. If a partially qualified file name is used, the syntax must conform to that of the file system selected.

If the file system is a Guardian file system, partially qualified file names are expanded using the Guardian defaults specified in the SYSTEM and VOLUME commands. For example. if the current system is \CALIF and the current volume is $DATA.FTRAN, the file alpha.beta is expanded to \CALIF.$DATA.ALPHA.BETA.

If the file system is an OSS file system, partially qualified names are expanded using the OSS default directory specified in the CD command. For example, if the current directory is /usr/guest, the file alpha.beta is expanded to /usr/guest/alpha/beta. The default is Guardian.

`FIXUPS {ON | OFF}`

specifies whether Binder fixes code and data references in the object file. If you select OFF, the object file cannot be run without first being reprocessed by Binder. The default is ON.

`IMPORT LIBRARY` *`library-file-name`*

specifies the use of an SRL *`library-file-name`* as a user library to be used during binding. The library specified must be a pre-existing instance of the user

library, that contains the library procedures and all the library variables referenced by the application. This command can only be used when you are building an application that will be linked to an SRL user library at run-time.

The imported library provides Binder with the recommended sizes to reserve for library data in the BELOW64, PRIMARY, SECONDARY, and EXTENDED data areas. It also provides the list of variables exported from the user library.

This command automatically turns on the RUNNABLE option and sets the default run-time library to *library-file-name*. If you want to specify a different run-time library, specify the SET LIBRARY command after the IMPORT command.

Specify this command anytime before the BUILD command. When the application is built, Binder compares the user library exported data block list with the current user code data block list. A warning is issued if any block definitions are not compatible.

Procedures specified in the SRL file are used only if they cannot be found in the other libraries specified in the SELECT SEARCH command.

The SRL must reside in the Guardian file system. You may specify an IMPORT library in the OSS file system. In this case, Binder uses the specified import library as a model for building, but it does not set the run-time library in the object file. The library can be set to a Guardian name using the SET LIBRARY command or the /LIB/ option in the TACL RUN command.

`LIST` *`listing-option`*

specifies the printed output for building the target file as one of the following:

```
{ALPHA {ON | OFF}}
{LOC {ON | OFF}}
{XREF {ON | OFF}}
{* {ON | OFF}}
```

`ALPHA {ON | OFF}`

specifies whether Binder produces a load map in alphabetic order. The map lists names and addresses for code blocks and data blocks, and also lists source file information for each code block. The default is ON.

`LOC {ON | OFF}`

specifies whether Binder produces a load map in location order. The map lists names and addresses for code blocks and data blocks, and also lists source file information for each code block or data block. The default is OFF.

`XREF {ON | OFF}`

specifies whether Binder produces a cross-reference list of entry points and data blocks. The default is OFF.

`* {ON | OFF}`

specifies whether you select all list options.

`OMIT` *`entry-name`*

specifies an additional entry point name for the omit list. The default is an empty omit list. Omit lists are described in Section 4, Object File Structure. Note that Binder omits all entry points belonging to a particular code block if any one of them appears on the omit list. For more information, see Considerations on page 3-56.

`REFER` *`refer-pair`*

specifies a pair of entry point names to be added to the refer list. The format for *`refer-pair`* is:

`entry-name``1 TO` *`entry-name`*`2`

The default is an empty refer list. See Considerations on page 3-56.

`RUNNABLE OBJECT {ON | OFF}`

specifies whether Binder should check at build time that the object file is runnable or not. For the file to be runnable, the following conditions must be true:

- The undefined data list must be empty.
- There must be a main procedure in the object file.
- Fixups must be applied to the object file.
- All function pointers must be resolved.

Binder cannot detect all errors; however, if it detects an error, it issues a warning message and continues to build the target file.

The default is OFF.

`SATISFY {ON | OFF}`

specifies whether Binder attempts resolution of remaining unresolved external references for entry points when you issue a BUILD or SATISFY command. Data block references are not affected by SATISFY. OFF suppresses automatic resolution; the default is ON.

`SEARCH` *`file-name`*

is a file name to add to the search list. The default is an empty search list.

`WARNINGS {ON | OFF}`

specifies whether warning messages be sent to the output file. The default is WARNINGS ON.

Using CHECK PARAMETER

- Binder provides parameter and return-value checking by matching parameters and return values types between called and calling routines. Because each language has its own set of data types, the matching of parameters and return-value types between languages cannot be exact. To reduce the number of extraneous Binder warnings generated when you bind mixed-language programs, Binder provides five levels of checking:

ON	Formal and actual parameters must be the same size, type, and mode (passed by value or by reference). Return values are not checked.
STRICT	STRICT is the same as ON.
STRONG	Interlanguage calls are checked as LENIENT. Calls between routines compiled from the same language are checked as STRICT.
LENIENT	Parameters and return values are viewed as members of classes. Corresponding formal and actual parameters and corresponding return values must belong to the same class.
OFF	No parameter type and return-value checking.

  Use the STRICT setting for single-language programs; use the STRONG setting for mixed-language programs.

- Binder groups parameters into these classes:
  - Two-byte scalar, passed by value.
  - Four-byte integer scalar, passed by value.
  - Four-byte real scalar, passed by value.
  - Eight-byte integer scalar, passed by value.
  - Eight-byte real scalar, passed by value.
  - Byte address.
  - Word address.
  - Extended address.
  - Two-byte procedure parameter.
  - Four-byte procedure parameter.

  Note that only the C language defines the passing of structured parameters by value. Such parameters cannot match parameters in any other language.

- Binder groups return values (values returned on the stack) into these classes:
  - Two-byte scalar.
  - Four-byte scalar.
  - Four-byte real scalar.
  - Eight-byte scalar.
  - Eight-byte real scalar.
  - No return type.

  Binder provides language consistency checking by making sure that callers specify the correct language for called routines. If a caller explicitly states that the language of a called routine is unspecified, Binder does not perform this check.

  If one caller specifies one language, and a subsequent caller specifies a different language, and neither of them has been resolved, Binder issues the message:

  ```
  **** WARNING 149 **** Referencing procedures do not agree
  on the language of procedure procedure-name.
  ```

  After it issues the message, Binder must determine which language to use for subsequent checking. If either caller is written in the same language as the called routine, Binder uses that language. If both callers specify languages other than that of the called routine, Binder selects the language of one of the caller routines. Binder makes this selection in the following order: COBOL85, FORTRAN, TAL, C, Pascal. For example, if one caller specifies C and another caller specifies COBOL85, Binder selects COBOL85 for the called routine.

  Once Binder has determined the language of a called routine, and a caller specifies a different language, Binder issues the message:

  ```
  **** WARNING 148 **** Referencing procedures claim
  that procedure procedure-name is written in a
  different language.
  ```

  If Binder issues these messages during a bind session, examine calls to the named procedure in your source code to make sure they specify the correct language. Note that two routines in the same object file cannot have the same name, even if they are written in different languages.

Considerations

- The SELECT command accepts OSS pathnames.
- OMIT does not remove an entry point name from the include list.You cannot use OMIT for nested code blocks.
- REFER works only for entry point references that are not present; it does not apply to entry points already on the include list.

- You cannot use REFER for nested code blocks (blocks with a lexical level greater than one).

Examples

The following examples illustrate the syntax of the SELECT command.

- This SELECT command allows a gap before the 32K boundary and suppresses entry point name resolution.

```
@SELECT COMPACT OFF, SATISFY OFF
```

- This SELECT command adds three file names to the search list and selects all error-checking options.

```
@SELECT SEARCH (obj1, obj2,obj3), CHECK * ON
```

- This SELECT command adds two pairs of entry names to the refer list and one entry name to the omit list.

```
@SELECT REFER (b1 TO b4, b2 TO b3), OMIT b5
```

SET Command

The SET command specifies attribute values to be associated with the target file. You can use the BUILD command to override attribute values or the RESET command to reset them to their default values.

```
SET { set-param [ , set-param ]... }
```

set-param

can be any of the following:

```
{DATA} value [PAGES | WORDS | BYTES]
{EXTENDSTACK}
{STACK}
HEAP value [PAGES | WORDS | BYTES]
HEAP STATISTICS {ON | OFF}
HIGHPIN {ON | OFF}
HIGHREQUESTERS {ON | OFF}
IMPORT DATA variable-name-list
INSPECT {ON | OFF}
LARGESTACK value [PAGES | WORDS | BYTES]
LIBRARY file-name
```

LIKE *file-name*

MISALIGN { FAIL| NOROUND | SYSDEFAULT}

PEP *value*

PFS *value* [PAGES | WORDS | BYTES]

RESERVE {BELOW64|PRIMARY|SECONDARY|EXTENDED} [+|-] nb

RUNNAMED {ON | OFF}

SAVEABEND {ON | OFF}

SUBTYPE *number*

SYMBOLS {ON | OFF}

SYSTYPE {OSS | GUARDIAN}

TARGET [TNS | TNS/R | TNS/E | ANY]

USERLIBRARY {ON | OFF}

USER BUFFER {ON | OFF}

DATA *value* [PAGES | WORDS | BYTES]

specifies the total amount of data space Binder allocates for the target file.

By default, Binder allocates the larger of the following two values: the maximum number of data pages in any of the files from which data is included or the number of pages needed to hold all the data blocks plus an estimate of the stack space needed for local storage. By using the DATA option you override Binder's default estimate.

For *value* you can enter either a decimal value or an octal value (preceded by %). If the value you specify in *value* is less than the amount of data needed to build the object file, Binder issues a warning.

To specify a unit of value, enter PAGES, WORDS, or BYTES. If you do not specify a unit, the default unit is PAGES. One PAGE is 1024 words.

EXTENDSTACK *value* [PAGES | WORDS | BYTES]

specifies the number of pages, words, or bytes you want to add to the Binder estimate of stack space for local storage. Binder increases its total allocation of space by the amount you specify.

You can specify either a decimal *value* or an octal *value* (preceded by %). The default unit for value is PAGES. One PAGE is 1024 WORDS.

STACK *value* [PAGES | WORDS | BYTES]

replaces the estimate of stack space for local storage computed by Binder with the value you specify. The number of pages, words, or bytes you specify plus

Binder's estimate of the space required for global data blocks is the total number of data pages allocated (one PAGE is 1024 WORDS). The default is the space Binder estimates for local storage.

You can specify either a decimal *value* or an octal *value* (preceded by %). The default unit for *value* is PAGES.

If you specify an odd number of bytes, Binder rounds up the value to an even value.

`HEAP` *value* `[PAGES | WORDS | BYTES]`

sets the maximum size of the heap used in C and Pascal. The default is 0.

You can specify *value* in pages, words, or bytes. The default unit for value is PAGES.

The specified *value* overrides any other value Binder has encountered for the heap size. If you omit this entry, Binder uses the largest #HEAP block.

`HEAP STATISTICS {ON | OFF}`

specifies whether a Pascal program collects heap statistics reported by calls to the supplied routine HEAPUSED. The default for C-series Pascal programs is ON; the default for D-series Pascal programs is OFF. You must set HEAP STATISTICS ON if your D-series Pascal program calls the HEAPUSED routine.

`HIGHPIN {ON | OFF}`

specifies whether an object file can run at a high process identification number (PIN), if the process creation request allows it and a high PIN is available. The default is HIGHPIN OFF.

`HIGHREQUESTERS {ON | OFF}`

specifies whether an object file can support high process identification number (PIN) requesters. The default is HIGHREQUESTERS OFF.

`IMPORT DATA` *variable-name-list*

specifies a list of variables to be imported. Use the SET IMPORT command to specify a list of data blocks to be imported from the SRL when building an SRL application. This command can be repeated, appending to the list of imported data blocks. It is optional for C-coded applications.

If the application is coded in C, imported data blocks are declared using the keyword extern. The data block is not defined within the application codefile, and Binder resolves the external references to the library. The IMPORT command is provided mainly for TAL users. In TAL, no extern designation exists. The IMPORT command instructs Binder to remove specific data blocks from the application.

`INSPECT {ON | OFF}`

specifies whether the Inspect program is chosen for debugging when you execute the target file. The default is OFF; that is, the Debug program is used. INSPECT OFF automatically causes Binder to set SAVEABEND OFF. You use the command interpreter SET INSPECT or RUN commands to override the INSPECT specification.

`LARGESTACK` *`value`* `[PAGES | WORDS | BYTES]`

sets the size of the $EXTENDED#STACK data block used in TAL. *`value`* can be specified in pages, words, or bytes. The default unit is PAGES.

The specified *`value`* overrides any other value Binder has encountered for this data block. If you omit this entry, Binder uses the largest $EXTENDED#STACK block.

`LIBRARY` *`file-name`*

specifies the name of the user library to associate with the object file at run time. You can override file-name at run time by using the LIB parameter in the command interpreter RUN command. This option does not support OSS pathnames. A Guardian filename must be specified. The default is no user library.

`LIKE` *`file-name`*

specifies that the DATA, INSPECT, LIBRARY, and SAVEABEND attributes for the target object file are to be identical to those of a specified object file.

`MISALIGN { FAIL | NOROUND | SYSDEFAULT}`

`MISALIGN FAIL` causes the system to generate instruction failure interrupts, which will be reported as trap 1 (INSTRUCTIONFAILURE) in a Guardian process, or a signal 4 (SIGILL) in an OSS process.

`MISALIGN NOROUND` allows all valid TNS programs and also erroneously-coded TNS programs to run, but not necessarily with the same results as on some prior machines. This option causes the system to complete the operation using the operand’s “natural” (unrounded) address.

`MISALIGN SYSDEFAULT` allows application of the current TNS Misalign policy. This codefile attribute is ignored when executing on TNS/R systems.

`PEP` *`value`*

specifies the size of the procedure entry point (PEP) table to be allocated for the target file. The default value is the minimum size necessary for the number of entry points in the target file. value can be any integer in the range 0 through 512. You can specify value either in decimal or in octal (preceded by %).

`PFS value [PAGES | WORDS | BYTES]`

specifies the size of the process file segment to be allocated in the target file. *value* can be from 64 to 512 pages (65,536 to 533,504 words). The default unit of size is PAGES. Binder rounds byte values up to the nearest page.

`RESERVE`

specifies how much space to reserve in the application's global data space for library variables. Binder must reserve at least enough space to accommodate the current library's global variables. The SET RESERVE command can be used to specify absolute maximum amount of space or an additional amount of space.

`BELOW64`

reserves space in the first 64 words of primary data. COBOL and TAL code allocate blocks in this area for use with the LWXX and SWXX instructions. These instructions provide efficient access to data in extended memory through pointers in the first 64 words of primary data. For TAL code, the INHIBITXX directive can be specified to avoid depending on this limited data space.

`PRIMARY`

reserves space in the first 256 words of primary data. This space is needed for directly addressed TAL variables, or C variables that must be accessible to TAL.

`SECONDARY`

reserves space in the first 256 words of secondary data. All scalar or pointer C variables that do not need to be accessed from TAL are allocated here.

`EXTENDED`

reserves space at the beginning of the extended segment. Most arrays, structures, and string constants are allocated here.

`[+|-] nb`

specifies the positive or negative number of bytes of space to reserve in the application's global data space. When an absolute amount is specified, it overrides any previous setting for that data area.

For more information on reserving space, see Section 6, User Libraries.

`RUNNAMED {ON | OFF}`

specifies whether an object file runs as a named process, even if no RUN command NAME parameter is specified. The default is RUNNAMED OFF.

`SAVEABEND {ON | OFF}`

specifies whether to create a save file if the process terminates abnormally during execution. Binder automatically sets INSPECT ON if SAVEABEND is ON. The default is OFF.

`SUBTYPE` *`number`*

sets the process subtype on input object files that contain a MAIN procedure. *number* is in the range of 0 through 63. Note, however, that values in the range of 1 through 47 are defined by Hewlett-Packard; values in the range of 48 through 63 can be defined by the user. The default value is 0.

BIND writes the specified *number* into the object file header. When you run the program, the operating system assigns to it the process subtype for an object file.

If you specify a SET SUBTYPE command before the MAIN procedure is added from an object file (with an ADD or SELECT SEARCH command, for example), Binder uses the subtype of the MAIN procedure instead of the specified subtype.

Add the MAIN procedure and then specify the SET SUBTYPE command if you need to set the subtype.

This option does not work on object files that do not have the MAIN attribute. To change the subtype on a procedure other than a main procedure, use the CHANGE SUBTYPE command.

`SYMBOLS {ON | OFF}`

specifies whether to retain Inspect symbol tables from the object files in the target file. The default is ON.

`SYSTYPE {OSS | GUARDIAN}`

specifies whether the target execution environment of an object file is the OSS environment or the Guardian environment. The default value is Guardian.

`TARGET [TNS | TNS/R | TNS/E | ANY]`

specifies the target processor that the target file runs on.

`TARGET TNS` specifies that the target file runs on a TNS processor only.
`TARGET TNS/R` specifies that the target file runs on a TNS/R processor only.
`TARGET TNS/E` specifies that the target file runs on a TNS/E processor only.
`TARGET ANY` specifies that the target file can run on either a TNS processor or a TNS/R processor.

By default, the target processor is unspecified. In this case, the target file can run on any processor. For more information, see Considerations on page 3-63.

`USERLIBRARY {ON | OFF}`

specifies the maximum number of code segments in an object file built by Binder.

- In a TNS system, the attribute specifies whether an object file is built with a maximum of 16 or 32 code segments. If the file contains 17 to 32 code segments, the process creation allocates the first 16 code segments in the user code space and the remaining in the user library space. You cannot run a program file that has more than 16 code segments by using an external user library file. When USERLIBRARY is set to ON, the object file is built with 32 code segments. When USERLIBRARY is OFF, the Binder builds using a maximum of 16 code segments.
- In a TNS/R system or a TNS/E system, the attribute specifies whether an object file is built with a maximum of 32 or 64 code segments. If the file contains 33 to 64 code segments, the first 32 code segments are allocated in the user code space and the remaining in the user library space. You cannot run a program file that has more than 32 code segments by using an external user library file. When USERLIBRARY is set to ON, the object file is built with 64 code segments. When USERLIBRARY is OFF, the Binder builds using a maximum of 32 code segments.

`USER BUFFER {ON | OFF}`

By default the USER BUFFER Flag will be set to OFF while creating an object file.

Considerations

- The SET command accepts OSS pathnames, except for the library option.
- Binder determines the target processor from the input object files. Table 3-3 shows the resulting Binder setting for various combinations of target processor settings.

Table 3-3. Resulting Target Processor Type

File 1 Target Type	File 2 Target Type	Resulting Binder Setting
Unspecified	Either unspecified or specified as any one of the three SET TARGET options	Setting of File 2
TNS	Unspecified or specified as either TNS or ANY	TNS
TNS	TNS/R or TNS/E	Error
TNS/R	Unspecified or specified as either TNS/R or ANY	TNS/R
TNS/R	TNS	Error

Table 3-3. Resulting Target Processor Type

File 1 Target Type	File 2 Target Type	Resulting Binder Setting
TNS/E	Unspecified or specified as either TNS/E or ANY	TNS/E
TNS/E	TNS	Error
ANY	Specified as any one of the SET TARGET options	Setting of File 2

If you try to bind two object files that contain conflicting TARGET attributes, Binder stops the current operation and issues the following error message:

```
****ERROR 165 **** TARGET types conflict:
entry point entry-name from file is type1, the current
setting is type2
```

If you enter the command SET TARGET ANY, and then later add in a procedure whose setting is TARGET TNS, TARGET TNS/R or TARGET TNS/E, Binder changes the target processor setting appropriately and issues the following warning message:

```
**** Warning 166 **** SET attribute has been changed:
from TARGET ANY to TARGET type
```

type can be TNS, TNS/R or TNS/E.

- For the DATA, EXTENDSTACK, STACK, HEAP, and LARGESTACK options, if you enter an odd number of bytes, Binder rounds up *value* to an even value.
- You can write process subtypes only into object files that contain a MAIN procedure.
- You can set only one of the DATA, STACK, or EXTENDSTACK parameters at a time. Each successive SET command specifying one of these parameters overrides the previous specification. Note the distinctions among these parameters:
  - DATA specifies the total amount of data space allocated for data blocks and local storage. This specification overrides Binder's estimate for the total amount of data space.
  - STACK specifies the amount of stack space allocated for local storage. This specification overrides Binder's estimate for local storage. Binder takes your specification for local storage, adds it to its space needed for these data blocks, and then computes total data space required.
  - EXTENDSTACK overrides Binder's estimate for local storage by specifying an addition to it. Binder increases its allocation of local storage by the amount of your addition and then adds together local storage and data space required for data blocks to compute total data space required.

- Extended data block sizes for all languages are set at compile time, and Binder collects all of these blocks into one extended segment that is automatically allocated at run time. SET DATA does not affect the size of this extended segment.
- Once set with the SET PFS command, the default process file segment size is 128 pages. At this point, entering a RESET command causes Binder to set the PFS size to 128 pages.
- If you do not set the size of the PFS, Binder uses the largest size encountered in the input object files when building the target file. Binder treats files created with a previous version of Binder as if they have a PFS size of 128 pages.

  The PFS value can be from 64 to 512 pages. If you specify a value outside of this range, Binder returns the following error message:

  ```
  Illegal PFS size. Legal range is 64 to 512 pages.
  ```
- The HIGHREQUESTERS ON attribute specifies that an object file is allowed to support requests from processes running at high PINs. It does not specify that an object file meets the conditions to support requests from processes running at high PINs.

Examples

The following examples illustrate the syntax of the SET command.

- The following command specifies that a save file is to be created if a process terminates abnormally when the target file is executed:

  ```
  @SET SAVEABEND ON
  ```
- The following example shows that two or more set parameters can be set in a single command. In this case the Inspect symbol tables from the object files are not to be retained in the target file, and the user library LIBFILE is to be associated with the target file at run time.

  ```
  @SET SYMBOLS OFF, LIBRARY libfile
  ```

Running Processes at a High PIN

The HIGHPIN ON attribute specifies that an object file is allowed to run at a high PIN. It does not specify that an object file meets the conditions to run at a high PIN.

When the operating system creates a process, it assigns a process identification number (PIN) to the process. D-series systems support the following ranges of PINs:

Low-PIN range	0 through 254
High-PIN range	256 through the maximum number supported for the processor

To run an object file at high PIN from the TACL prompt, the following conditions must be met:

- Your processor is configured for more than 256 process control blocks (PCBs).
- High pins are available in your processor.
- Your object file and user library, if any, have the HIGHPIN attribute set.

The TACL HIGHPIN built-in variable or the HIGHPIN run-time parameter is set.

If the HIGHPIN attribute of the object file is set, the operating system assigns a high PIN, if available. If no high PINs are available, the operating system assigns a low PIN.

You can set the HIGHPIN attribute of an object file either:

- During compilation by using the HIGHPIN directive
- After compilation by using a Binder command

If the above conditions are met, your object file can create another process to run at high PIN by specifying the PROCESS_CREATE_ system procedure with create-options bit 15 set to 0 and bit 10 set to 1.

The following sequence of examples show how to run an object file at high PIN from the TACL prompt. The examples show how to check your processor configuration and high-PIN availability, set the HIGHPIN attribute, and override the TACL HIGHPIN setting if it is off.

1. To check the number of PCBs configured in your processor and to see if high PINs are available, run the Peek product. For example, if you want to run your object file on processor 1:

   ```
   PEEK / CPU 1 /
   ```

   The following display excerpt shows example values for the information you need to check:

   ```
               ... CURRENT USAGE   # CONFIGURED...
   PCB         127: 48             255: 244
   ```

   The processor is configured for high PINS if the sum of the two values displayed for PCBs under # CONFIGURED is 256 or greater.

   The processor has high PINs available if the righthand value under CURRENT USAGE is less than the righthand value under # CONFIGURED.

2. You can set the HIGHPIN attribute of an object file during compilation by including the HIGHPIN directive in the compilation command:

   ```
   TAL /IN talsrc, OUT $S.#tallst, NOWAIT/ talobj; HIGHPIN
   ```

3. Alternatively, you can set the HIGHPIN attribute of an object file after compilation by typing the following Binder command:

   ```
   BIND CHANGE HIGHPIN ON IN talobj
   ```

4. Before you run the object file, you can check the current setting of the TACL HIGHPIN built-in variable by typing:

```
#HIGHPIN
```

5. If #HIGHPIN returns a NO value, you can set the HIGHPIN run-time parameter (and run your object file at high PIN):

```
RUN talobj / HIGHPIN ON /
```

6. If #HIGHPIN returns a YES value, you can simply run your object file at high PIN:

```
RUN talobj
```

Refer to the *Guardian Operating System Application Conversion Guide* for additional details on writing processes that run at a high PIN.

SHOW Command

The SHOW command displays the current values for the following:

- Current file.
- Set of modifications established by the MODIFY command.
- SELECT command parameters.
- Attributes of an object file.
- SET command parameters.

After a BUILD command executes, Binder resets SELECT, SET, FILE, and all lists to the default states. SHOW SET *attribute* then displays information about the constructed target file.

```
SHOW [ / OUT file-name / ]
{ AXCEL ENABLE [ FROM file-name ]   }
{ FILE                              }
{ IMPORT                            }
{ INFO [ FROM file-name ]           }
{ MODIFY                            }
{ OCA ENABLE [ FROM file-name ]     }
{ RESERVE                           }
{ SELECT                            }
{ select-param                      }
{ SET attribute [ FROM file-name ]  }
{ SET                               }
{ set-param                         }
```

OUT *file-name*

directs the output listing to the specified file. For additional information, see OUT Command on page 3-39.

`AXCEL ENABLE [FROM` *`file-name`*`]`

displays the current state, either ON or OFF, of the AXCEL ENABLE attribute for the current file or file specified in the FROM clause. You can set this attribute with the CHANGE command. OSS pathnames are accepted.

`FILE`

displays the name of the current file.

`IMPORT`

displays the list of imported data blocks specified for the current session. This command only lists the set of variables explicitly specified in the SET IMPORT command. The complete import list is not determined until build time.

`INFO [FROM` *`file-name`*`]`

displays attributes (number of data pages, process subtype, number of code segments and file segments, system type, Accelerator information, Binder information, and debugging information) associated with the current Guardian file or OSS pathname, or the Guardian file or OSS pathname specified in the FROM clause. The output will include Itanium instruction information if the program has been translated by OCA and will show the MISALIGN attribute of the object.

`MODIFY`

displays the current set of modifications established by the MODIFY command. If code has been modified, the values are displayed in ICODE as well as octal.

`OCA ENABLE [ FROM` *`file-name`* `]`

This command shows whether the named TNS object file is allowed to be translated by OCA, and, if an Itanium instruction region exists, whether its Itanium instructions can be executed in a translated mode of execution.

`RESERVE`

displays the reserve settings for each of the four data areas: BELOW64, PRIMARY, SECONDARY, and EXTENDED.

`SELECT`

displays the current values for all the SELECT command parameters.

`select-param`

displays one of the following SELECT command parameters:

`CHECK`	`SATISFY`
`REFER`	`FILESYS`
`COMPACT`	`FIXUPS`

RUNNABLE OBJECT	SEARCH
COMPRESS DATA	LIST
WARNINGS	OMIT

SET *attribute* [FROM *file-name*]

displays one, or all, of the attributes associated with the specified object file. *attribute* is one of the following:

*	SAVEABEND
RESERVE	HIGHREQUESTERS
DATAPAGES	SUBTYPE
RUNNAMED	IMPORT
HIGHPIN	SYMBOLS
INSPECT	SYSTYPE
LIBRARY	TARGET
PFS	TIMESTAMP

If you enter * as the value of attribute, the values of all the attributes are displayed.

FROM file-name specifies the object file whose attributes are to be displayed. If FROM file-name is omitted, the current file is used. OSS pathnames are accepted.

SET

displays the current values for all SET parameters.

set-param

displays one of the following SET command parameters:

DATA	PEP
EXTENDSTACK	PFS
HEAP	RUNNAMED
HEAP STATISTICS	SAVEABEND
HIGHPIN	STACK
HIGHREQUESTERS	SUBTYPE
INSPECT	SYMBOLS
LARGESTACK	SYSTYPE
LIBRARY	TARGET
LIKE	USERLIBRARY
MISALIGN	

Considerations

- SHOW SET PFS, SHOW PFS, and SHOW SET display the value of the PFS with one of the following units of measure: PAGES, WORDS, or BYTES. If the value is not a round number of pages, the value will be displayed in WORDS.
- SHOW SET `attribute` works on stripped files (files without a Binder region).
-
- Note the distinctions among the SET `attribute`, SET, and `set-param` options of the SHOW command.
  - SHOW SET `attribute` displays the value of the specified attribute of the current object file or object file specified in FROM `file-name`.
  - SHOW SET * displays the value of all the attributes associated with the current object file or the object file specified in FROM `file-name`.
  - SHOW SET displays the values of all the parameters established by the SET command.
  - SHOW `set-param` displays the value of a particular parameter established by the SET command.
  - SHOW SET MISALIGN displays the misalign attribute associated with the specified object file.

Examples

- The following command displays the name of the current file:

```
@SHOW FILE

FILE                 \SALE.$EUROPE.PARIS.BROKERS
```

- The following command displays the value of the DATA parameter established by the SET command:

```
@SHOW DATA

DATA                (0 PAGES)
```

- The following command displays the values of all the parameters established by the SELECT command.

```
@SHOW SELECT
CHECK            BLOCK ON, LIBRARY OFF, PARAMETER STRICT
COMPACT          ON
COMPRESS DATA
FILESYS          GUARDIAN
FIXUPS           ON
LIST             ALPHA ON, LOC OFF, XREF OFF
OMIT             0 ENTRIES
REFER             0 ENTRIES
RUNNABLE OBJECT  OFF
SATISFY           OFF
SEARCH            0 ENTRIES
WARNINGS          ON
```

- The following command displays the values of all parameters established by the SET command:

```
@SHOW SET
DATA                (64 PAGES)
EXTENDSTACK
HEAP
HEAP STATISTICS     OFF
HIGHPIN             OFF
HIGHREQUESTERS      OFF
INSPECT             ON
LARGESTACK
LIBRARY
PEP
PFS
RUNNAMED           OFF
SAVEABEND          OFF
STACK
SUBTYPE
SYMBOLS            ON
SYSTYPE GUARDIAN
TARGET ANY
USERLIBRARY        OFF
```

- The following command displays the value of the COMPACT parameter of the SELECT command:

```
@SHOW COMPACT
COMPACT      ON
```

- The following command displays the values of all the attributes associated with the current object file:

```
@SHOW SET *
TIMESTAMP      1994-10-24 11:51:06
LIBRARY        NOT USED
DATAPAGES      2
HIGHPIN        OFF
HIGHREQUESTERS OFF
INSPECT        ON
RUNNAMED       OFF
SAVEABEND      OFF
SYMBOLS        OFF
SEGMENTS       1
SUBTYPE        0
PFS            0 WORDS
TARGET         UNSPECIFIED
```

- The command SHOW INFO FROM *file-name* produces a listing such as the following for accelerated files:

```
@SHOW INFO FROM myfile
Filename: \NCAL.$SANTA.CLARA.MYFILE
       General Information
       Binder region: YES
    Binder timestamp: 1994-03-01 16:38:53:23
          Data pages: 64
            Debugger: INSPECT
      Inspect region: YES
     Process subtype: 0
Program file segment: 1024 WORDS
      Highrequesters: OFF
            Runnamed: OFF
             Highpin: OFF
          Saveabend: ON
           Segments: 3
             Target: TNS/R
Accelerator Information
Accelerated Execution: ENABLED
         Optimization: PROCDEBUG
       Global options: ATOMIC_OFF, INHERITSCC_ON,
                       OVTRAP_ON, SAFEALIASINGRULES_OFF,
                       TRUNCATEINDEXING_ON
                       Timestamp: 1994-03-01 14:24:56:14
                       Version: 1994-03-01 12:04:05:16
```

- The command SHOW INFO FROM *file-name* produces a listing such as the following for D-series object files:

```
@SHOW INFO FROM myfile
Object File: \RUSTY.$USER.TEST.MYFILE
        General Information
       Binder Region: YES
    Binder Timestamp: 1994-10-24 11:51:06.30
          Data Pages: 2
            Debugger: INSPECT
      Inspect Region: NO
     Process Subtype: 0
 Program File Segment: 0 WORDS
      Highrequesters: OFF
            Runnamed: ON
             Highpin: ON
           Saveabend: OFF
            Segments: 1
              Target: UNSPECIFIED
```

STRIP Command

The STRIP command removes the Binder, Inspect, and Accelerator regions from the named object file. STRIP can also selectively remove the Itanium instruction region from the named object file.

Note that STRIP modifies the named file; Binder does not copy the file to a target file. If you need Binder and Inspect tables for future analysis, copy the file to another location.

```
STRIP file-name [ , SYMBOLS | , AXCEL| , IPF / OCA ]
```

file-name

specifies the Guardian disk file name or OSS pathname of an object file whose Binder region and Inspect region (if any) are to be deleted. STRIP file-name leaves the Accelerator region in the file.

SYMBOLS

strips the Inspect region containing symbols tables from the object file, leaving the Binder, Accelerator and IPF regions.

`AXCEL`

strips the Accelerator region (containing TNS/R code) from the object file, leaving the Binder and Inspect regions.

`IPF`

strips the Itanium region (containing TNS/E code) from the object file, leaving the Binder and Inspect regions. Use of IPF or OCA keywords produce an equivalent result.

Considerations

- To strip all regions from a specified file, enter three commands `STRIP` *`file-name`* then `STRIP` *`file-name`*`, AXCEL` then `STRIP`*`file-name`*`, IPF`
- The named object file can be stripped of its Binder, Inspect, and Accelerator regions whether it contains one code segment or multiple code segments.
- The named object file can be stripped whether or not it contains extended data blocks.
- After an object file is stripped of its Binder tables, you can use the CHANGE, LMAP, SHOW, and STRIP commands on it.
- After an object file is stripped, it can still be executed, and it can be debugged by low-level Inspect or Debug.
- If you want to produce a target file without Inspect symbol tables but do not want to strip the symbol tables from the input object file, issue the SET SYMBOLS OFF command before issuing the BUILD command. This procedure does not change the input file on disk. Binder copies the file to the target file, minus the Inspect symbol tables.

Examples

The following examples illustrate the syntax of the STRIP command.

- The following command strips both the Binder and Inspect regions from the named object file:

```
@strip cref.zcobext3
@
```

- The following command strips the Accelerator region from the named object file:

```
@strip cref.zcobext4, AXCEL
@
```

- The following two commands strip the Binder, Inspect, and Accelerator regions from the named object file:

```
@strip cref.zcobext5
@strip cref.zcobext5, AXCEL
@
```

SYSTEM Command

The SYSTEM command specifies the default node that Binder uses to expand file names.

```
SYSTEM [ node ]
```

node

is a node name.

Considerations

- If you specify a node name with the TACL SYSTEM command, Binder expands files using that node name in any subsequent Binder command that does not specify a node name.
- Conversely, if you reset the TACL SYSTEM command to its default setting, Binder expands files on your default node for any subsequent Binder command that does not specify a node name.

VERIFY Command

The VERIFY command compares the value of a code or data word in an object file with a value specified by the user. If the value in the object file is not identical to that specified, the BIND process terminates.

You can use the VERIFY command with nested code blocks (blocks with a lexical level greater than one); however, if you do not fully qualify the name, Binder uses the first block found in the file.

```
VERIFY { CODE block-name } [ verify-spec ] [ offset ] , value
       { DATA block-name }
```

CODE *block-name*

specifies the name of a code block containing a value to verify.

DATA *block-name*

specifies the name of a data block containing a value to verify.

verify-spec

is one of the following keywords that specifies the format for value:

```
ASCII

DECIMAL

HEX

OCTAL
```

The default is OCTAL.

offset

is the offset, in octal, from the base of the block used to compute the exact location of the word whose value Binder verifies against value. The default is the base of the block (offset 0).

value

is an expression that Binder verifies against the contents of the specified word. Enclose an ASCII value in quotation marks.

Considerations

You can use the VERIFY command during a noninteractive BIND session to check that the value of a code or data word corresponds to the value you expect. If the VERIFY command yields a discrepant value, Binder issues the fatal error message “Value specified in VERIFY command not equal to current value” and terminates.

Examples

The following examples illustrate the syntax of the VERIFY command.

```
@VERIFY CODE block-3, 17

@VERIFY DATA block ASCII 50, "ab"
```

VOLUME Command

The VOLUME command specifies the default volume and subvolume names that Binder uses in the expansion of Guardian file names.

```
VOLUME { $ volume               }
       { [ $ volume. ] subvol }
```

$ volume

is a volume name.

subvol

is a subvolume name.

Considerations

- The ENV command lists the Binder default volume and subvolume.
- The VOLUME command is only used when the file system is set to Guardian in the SELECT FILESYS command.

4 Object File Structure

This section discusses object file structure, including:

Topic	Page
Code Blocks, Entry Points, and Data Blocks	4-1
Object File Format	4-8

All Binder operations are performed on object files. The following discussions pertain to object file structure both for input files and target files.

Code Blocks, Entry Points, and Data Blocks

Code blocks and data blocks are the smallest independently located pieces of a program. Code blocks contain executable machine instructions, as well as some inline constant data. Data blocks contain only data.

Code blocks are all located in the read-only code segments of the program's user code space or the user library space. There can be 1 to 32 code segments, each containing up to 64K 16-bit words of code. Binder automatically determines where to put code blocks within the minimum number of segments, unless you manually override this determination with Binder commands.

Because code and library spaces are entirely read only, they are automatically shareable among all processes concurrently executing an object code file. In contrast, each process has its own private copy of the program's data space. The name, visibility, size, contents, and internal layout of a block are determined at compile time. A block's location in memory is determined at bind time. Each block's location in memory is independent of the locations assigned to other blocks. A major function of binding is to fix up or relocate all address references from one block to another after the locations of all blocks are known.

Code Blocks

A code block contains the executable machine code for a routine that is invoked through a procedure call (PCAL) or an external call (XCAL) instruction and the procedure entry point (PEP) table.

Code Block Names

The name of the code block is the same as that of the routine it contains. Thus, a code block can be a COBOL85 compilation unit; a Pascal main program, procedure, or function; a TAL PROC; a FORTRAN main program, subroutine, or function; or a C function. The name given to an unnamed FORTRAN main program is MAIN^. For more information on PCAL and XCAL instructions, see the appropriate description manual for your system.

Code block names are not case-sensitive except when the program includes C-coded routines. You can call a C routine from other languages only if the routine name in the C program is in all uppercase characters.

Code Block Attributes

Especially in TAL and Pascal, code block declarations can contain attributes that define execution or relocation characteristics. You can alter some attributes using the ALTER command. Once you have set the INTERRUPT, EXTENSIBLE, and VARIABLE attributes of a TAL code block or the EXTENSIBLE attribute of a Pascal program at compile time, however, you cannot change them.

The MAIN attribute applies to any of the languages. After compilation, however, you can alter the MAIN attribute for TAL and C procedures only.

You can explicitly set the attribute in the given language compiler if “yes” is shown in the table.

Table 4-1. Code Block Attributes

Attribute	C	COBOL85	FORTRAN	PASCAL	TAL	Alterable by Binder
MAIN	Yes	Yes	Yes	Yes	Yes	TAL, C
CALLABLE				Yes	Yes	TAL, Pascal
INTERRUPT					Yes	
PRIVILEGED				Yes	Yes	TAL, Pascal
RESIDENT				Yes	Yes	TAL, Pascal
VARIABLE					Yes	
EXTENSIBLE				Yes	Yes	

For descriptions of the code attributes, see the *TAL Reference Manual* discussion of procedure and subprocedure declarations.

Primary and Secondary Entry Points

Each code block has one primary entry point, accessed through PCAL or XCAL instructions. The name of the primary entry point is the same as the code block. The primary entry point is generally, but not always, at the first word of the code block.

FORTRAN and TAL allow routines to have zero or more secondary entry points, which are also accessed through PCAL instructions. Other languages do not use secondary entries. To a caller, a secondary entry point looks and acts like independent routines; it is sufficiently related to the primary entry point to share source code and machine code and cannot be positioned independently at bind time. Secondary entry points have distinct names and distinct starting locations within the code block.

FORTRAN arithmetic statement functions, TAL SUBPROCs, and COBOL85 PERFORM ranges are routines that are invoked through branch to subroutine (BSUB) instructions; these routines do not have their own code blocks. The code for such routines must be contained in the same block as their callers. The location of the routine's code within the local code block is determined at compile time. Binder knows nothing and does nothing about BSUB routines. Pascal and C do not use BSUB routines.

Routine Scope

The scope or visibility of a routine name, code block name, or entry point name can be public (global) or private. A routine's status depends on the language as follows:

- In FORTRAN and TAL, all user routines are public and are callable from anywhere in the whole program.
- In COBOL85, only user routines that are separately compiled are callable from anywhere in the whole program.
- In C and Pascal, some routines of a module are public, but others can be private to the module. Specifically, in C, public routines are called external and private routines are called static. In Pascal, public routines are explicitly exported from their module, whereas nonexported routines default to private.

A private routine, code block, or entry point is limited to a module or to its encompassing procedure, that is, a nested procedure. Public routine names must be unique across the whole program. Private routine names need only be unique within their scope. It is possible for private routines in different scopes to have the same name, or even for a private routine to have the same name as a public routine. You resolve ambiguities by prefixing the routine name with the name of its scope. This is called qualifying the routine name.

To call or reference a private routine, you can use its unqualified name if it is unique to the whole program. Otherwise, you must use the fully qualified name. For example, you would refer to a routine P private to module M as M.P; you would refer to a routine P private to public routine Q as Q.P. In Pascal and C, module names are not the program or module declaration, however, but the file name. You name routines private to a Pascal or C program by prefixing the routine name with the module scope name. The module scope name is the name of the module's primary source file.

Nested Routines

In Pascal and COBOL85, user routines can be textually nested within larger routines. In the other languages, you cannot nest user routines. Nested routines are private to the routine containing them.

You reference private routines by prefixing the inner routine name with the fully qualified name of the outer routine. Here is an example of a nested routine in Pascal:

- Routine P is directly nested within routine Q.

- Q is directly nested within outermost routine R.
- R is private to module M compiled from source file $VOLUME.SUBVOL.FNAME.

The full name for routine P is FNAME.R.Q.P. FNAME, the file name, and not M, the module name, is used. The fully qualified name must not be longer than 255 characters.

Routines textually nested within a Pascal main program are compiled as if they were not nested. The scope name qualifying these routines is the main-program module's primary source file, rather than the declared name of the main-program routine.

Data Blocks

A data block is a collection of statically allocated variables or constants. The compilers decide how many blocks are needed, how the blocks are accessed, what data to put in each block, and where to put it within the block. Binder determines where to put the entire block, within the address space of the process. The location or size of the block does not change during process execution. All data blocks go into the data space of the process, except for TAL read-only blocks, which reside in the read-only code space of the process.

Programs also have dynamically allocated variables, whose locations are not determined until the process is executing. These include routines' local variables residing in the stack, file buffers, and anonymous objects in C or Pascal's heap space. These variables are not part of any data block.

Binder must resolve data block references during binding. Therefore, you cannot create an object file until all the necessary data blocks have been compiled. An object file is not executable until all data block references have been resolved.

In addition to its contents and initial values, a data block has the following attributes:

- Name
- Scope (which routines can reference the block)
- Means of access (where it can be located)

The scope of a data block is either:

- Public to all modules, known to Binder as a common data block
- Private to one module or private to one code block, known to Binder as an own data block

Binder recognizes one other type of data block called a special data block. These blocks are generated either by a compiler or by Binder. Program control blocks such as the run-unit control block (RUCB) and the program-unit control block are special blocks.

Special block names are distinguished by having a pound sign (#) for at least one of the characters in the name. The data block output listings display the special block names. You do not normally use these names in Binder commands.

Note. Uninitialized data in extended data blocks is compressed out of the object file space. The operating system restores the extended data space to its original space before it was compressed when the file is run. This enhancement saves disk space for object files with uninitialized extended data space.

Common Data Blocks

You can reference a public or common data block from any module that declares it. The name of a common data block must be unique.

Each programming language has its own terminology for common data blocks.

- In FORTRAN, they are known as named or blank common blocks. (Blank common is named BLANK^ by the compiler.)
- In COBOL85, they are known as external records.
- In TAL, they are known as named or private blocks. (TAL's private block is treated as named, using the module's name. TAL variables declared before the first explicit BLOCK declaration are put into an implicit block named #GLOBAL.)
- C and Pascal do not have a construct for declaring a block containing several variables. Instead, every public variable is automatically treated as a block containing exactly one variable; the block is named after the variable it contains.
  - In C terminology, a public variable is described as an external, nonstatic variable (which is a misnomer, because all nonlocal variables are statically allocated).
  - In Pascal, a public variable is a variable that is explicitly exported from one module and imported into others.

A common data block can be owned by several modules, which means that they all attempt to specify the block's initial values. Make sure that all owners consistently specify the same initial values; otherwise, the results will be random. C and Pascal modules can distinguish between importing a public block, and owning and exporting the block. FORTRAN, TAL, and COBOL85 are unable to declare a public data block without also claiming to be its owner.

FORTRAN subprograms do not have to agree on the size of a particular public data block; Binder uses the largest of the specified sizes.

Own Data Blocks

You can reference a private or own data block only from the module or code block that owns it. Its name need not be unique. If the name is not unique, you must fully qualify it in Binder commands, by prefixing it with the module scope name or the fully qualified routine name.

Some compilers generate private data blocks to handle the collection of constants and all private, statically allocated variables used by the compilation unit. For C and Pascal, these data blocks are private to the module; for COBOL85, and FORTRAN, these data blocks are private to one routine. Each language uses a different terminology for private, statically allocated variables:

- In C, these are static, nonexternal variables.
- In COBOL85, these are nonexternal items in the Working Storage and Extended Storage sections.
- In FORTRAN, these are SAVE variables and variables that appear in DATA statements but not in common statements.
- In Pascal, these are nonexported, nonimported variables declared at the outermost level.
- TAL does not use this kind of data.

The HP NonStop architecture uses four different ways to address statically allocated data, with different tradeoffs in speed, capacity, or generality. These four ways create four categories of data blocks:

1. Directly addressable data residing in the first 256 words of the data space, accessed through small offsets from the G register.
2. Data indirectly accessed through 16-bit G-relative addresses, located anywhere below the stack, within the first 32K words of the data space.
3. Data indirectly accessed through 32-bit extended addresses, located anywhere within the data space.
4. P-relative read-only data residing in the local code segment, accessed through the P register (TAL only).

Each compiler has different strategies for supporting its language through a combination of these categories of static data.

A single block declaration at the source level can actually produce several data blocks at the machine level, putting some of the conceptual block's data into each of the four categories of storage. Consider the following TAL example:

```
BLOCK BLK;

INT      A;

INT     .B;

INT .EXT C;

INT         D[0:9];

INT        .E[0:99];

INT .EXT F[0:9999];
```

```
INT G[0:7] = 'P' := "abcdefgh";

END BLOCK;
```

At the conceptual level, this declares a single block named BLK, containing seven variables. This actually gets compiled as four related blocks:

BLK containing directly accessed items residing in primary global space:

- The value of the numeric variable A
- The value of the 16-bit pointer variable B
- The value of the 32-bit pointer variable C
- The contents of the directly-addressable array variable D
- A 16-bit pointer for the indirectly-addressed array variable E
- A 32-bit pointer for the indirectly-addressed array variable F

.BLK containing indirectly accessed items residing in secondary global space:

- The contents of the array variable E

$BLK containing indirectly accessed items residing in extended global space:

- The contents of the array variable F

G containing read-only data residing in code space:

- The contents of the array variable G

Binder derives the names of indirectly accessed blocks from the primary block's name by adding the prefix character "." for 16-bit addressing or "$" for 32-bit addressing.

Other languages use some subset of TAL's scheme of multiple blocks; refer to the appropriate language manual for additional information.

Separately compiled modules need not use the same technique for addressing data in public data blocks. For example, when accessing a block in secondary global named .BLK, TAL always requires a directly accessible block in primary named BLK containing pointers into .BLK. FORTRAN does not use BLK to access .BLK. And under the large memory model, C and Pascal also do not use BLK to access .BLK. If BLK happens to be present, for the sake of TAL modules, it is ignored by FORTRAN, C, and Pascal modules.

Separately compiled modules need not agree on what category of memory is required for a particular public data block. For example, TAL always requires that a block named BLK be in primary global, below G+256, so that it is directly addressable. But under the large memory model, C and Pascal do not require that BLK be directly addressable; any location in secondary global, below G+32768, will do. If these TAL and C or Pascal modules are bound together into one program, Binder places BLK in primary global to meet the most restrictive requirements on its placement.

You can modify separately compilable data blocks in an object file either by recompiling the block or by using the MODIFY command. Then, you can build a new object file with the corrected data block during an interactive BIND session. You do not need to recompile the entire program.

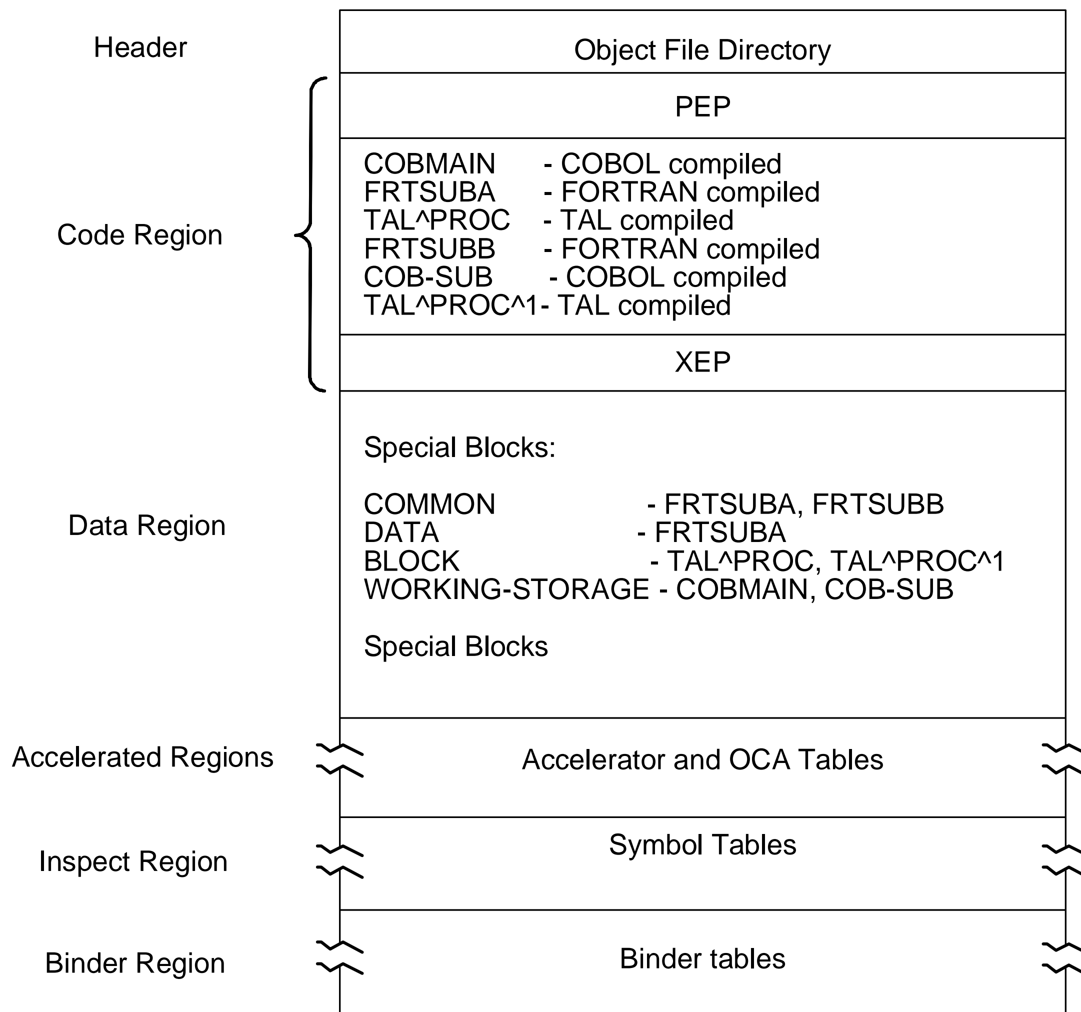
Object File Format

All object files have the same format, regardless of the number of code and data blocks.

Figure 4-1 shows an example object file made up of several blocks copied from different object files. For additional information on mixed-language binding, see Binding Mixed-Language Programs on page 2-13.

On disk, an object file can consist of a maximum of 16 extents. The header, code, and data areas must be in the first extent (0). You can use the additional extents (1 through 15) for Inspect symbol tables and Binder tables.

Figure 4-1. Example of the Binder Object File Format



Note. This figure shows a single code segment in the code region. Normally, many more code segments would appear.

Note. The object file format may change. New versions of Binder will accept all existing Binder object file formats; however, old versions of Binder will not accept new object file formats.

Header

The object file header is a block at offset zero containing pointers and descriptive information for other regions in the object file.

Code Region

The code region consists of consecutive pages of disk space, starting on a page boundary, in the object file. Binder output statistics give the exact number of pages to be allocated for the code area at run time.

Code region contents, in order, are:

1. PEP table
2. Global read-only arrays (TAL only)
3. Resident code blocks (TAL only)
4. Nonresident code blocks
5. XEP table

You can define the order of blocks that Binder uses to build the code region. In building the file, Binder separates resident code blocks from nonresident code blocks for you.

In the completed target file, nonresident code can come before resident code if compression of the target file requires it. Binder compresses the file contents by moving nonresident blocks that fit in the gap at 32K even if resident code blocks are placed above the 32K boundary. The SELECT COMPACT OFF command prevents Binder from performing any compression or filling the gap.

Multiple-Code-Segment Files

For small programs, Binder builds single-code-segment files. But if a program exceeds 64K words of code or 512 entry points, Binder builds its object file with multiple code segments. You can also create multiple-code-segment files explicitly by using the ADD SPACE command or the MOVE `entry-list` IN NEW SPACE command.

TAL Global Read-Only Arrays

Binder supports the concept of read-only global data arrays. Currently, the only language that supports global read-only arrays that are visible to Binder is TAL. The TAL term for read-only arrays is P-relative arrays. The name is derived from the fact that read-only arrays reside in the code area of the object file rather than the data area.

Although it can be confusing to find data residing in the code area, it makes sense when you realize that you can both read the data area and write to it, whereas you can only read the code area. Therefore, read-only data can safely reside in the code area, where you cannot accidentally overwrite it.

For single-code-segment object files, Binder places these arrays in the code region immediately following the PEP. For multiple-code-segment object files, Binder places these arrays with the first code block in each code segment that refers to them. They can therefore appear in more than one code segment.

If possible, Binder does not allow read-only string arrays to straddle the 32K boundary. Binder tries to place such an array above or below the boundary to avoid straddling it. If it cannot fit the array above or below the boundary, Binder places the array across the boundary and issues a warning message.

If a string P-relative array is below the 32K boundary and a procedure that refers to this array is above the boundary, Binder issues a warning message.

You can tell which data blocks are read only in the Binder load map by noticing the “C” that prefixes the base address of the data block. While the base addresses of the read-only data blocks in the load map are code segment addresses, Binder lists the read-only blocks either separately or with the regular data blocks. See Figure 2-6 for a sample listing of a read-only data block.

Using read-only data blocks in Binder commands is straightforward. Binder treats all read-only data blocks as DATA blocks, not CODE blocks. Thus, if you want to perform a Binder function on a read-only data block, you need only specify DATA when using a Binder command that distinguishes between code blocks and data blocks.

Binder specifies the base addresses and limits of read-only data blocks in the load map just as it does for regular data blocks. In the following example, the values specified to dump the first ten words of a regular data block and of a read-only data block are expressed in the same way:

```
DUMP DATA regular^data 0,10

DUMP DATA read^only^data 0,10
```

PEP and XEP Tables

The PEP (procedure entry point) and XEP (external entry point) tables are operating system tables.

The PEP table contains the entry point addresses for each code block in each code segment. The PEP is in the first page of each code segment.

The XEP table is in the last page of each code segment and contains an entry for each unresolved external reference. The operating system fills in this table at run time. These unresolved references can refer to entry points in the code or system library, the FORTRAN or COBOL85 run-time libraries, or to entry points in user library segments.

Data Region

Binder determines the minimum number of pages to allocate for the data area and reports the number in the output statistics. The data area starts on a page boundary.

Binder also determines the order for data blocks so you need not specify their order when you define a target file.

Accelerator Region

After you have run a run a file through the Accelerator, the Accelerator region contains the TNS/R version of the original TNS instructions. If you do not use the Accelerator, this region is a zero-length record. See the *Accelerator Manual* for information on the Accelerator.

OCA Region

After you run a file through the Object Code Accelerator (the OCA), the OCA region contains the TNS/E version of the original TNS instructions. If you do not use OCA, this region is a zero-length record. See the *Object Code Accelerator Manual* for information on OCA.

Inspect Region

Symbol tables in the Inspect region contain information on all symbols in blocks that were compiled with the SYMBOLS directive. Because you can turn SYMBOLS on and off for each procedure, an object file can contain symbol tables for some of the blocks and not for others.

The space required for symbol tables depends on program characteristics. Space requirements for the object file can increase significantly when data requirements are complex. During an interactive Binder session, you can specify whether to retain the symbol tables in the target file. You should retain symbol tables for blocks that are still in the development cycle if you plan to use Inspect high-level commands. See the STRIP Command on page 3-75

Once you delete the symbol tables, you can still use the LMAP command to display the load map of the stripped file. You can also use low-level Inspect commands and Debug commands whether symbol tables exist or not. If you need the symbol tables, you must recompile the file. For information on debugging programs, see the *Inspect Manual* and the *Debug Manual*. For information on LMAP, see LMAP Command on page 3-32

Binder Region

The Binder region contains a header and the following Binder tables:

- Procedure information table
- Entry point table
- Data block information table

You can strip Binder tables using the STRIP command. See STRIP Command on page 3-75

5 Binder Input and Output

This section describes the two stages of Binder operation: input and output. It contains the following topics:

Topic	Page
The Input Control Lists	5-1
The Target File	5-8

During the input stage, you instruct Binder how to build the target file by specifying the names of input object files and the code and data blocks within those files. You also specify any changes to the code blocks, data blocks, and references that Binder must follow to build its target file.

During the output stage, Binder builds the target file according to your instructions and places the new target object file on disk.

Compilers provide names for all code and data blocks that are unnamed in the source code. In turn, Binder receives from the compilers and interactive users the names of disk files to search for specific code and data blocks. When the Binder finds each block, it copies the code or data block into the target file. The input object file is not affected; it remains on disk in its original state.

Binder determines which code blocks should be included in the final executable program from among all the available code blocks contained in the input files (specified in ADD and SELECT SEARCH commands). Binder makes these determinations independently for each code block using the input control lists. If an input object code file has four routines—A, B, C, and D—Binder might determine that only B and C are needed in this application. Binder would then copy B and C into the new object code file and omit A and D.

Binder makes these determinations by tracking which routines are needed to satisfy direct or indirect calls from the main program. Routines that are called but not yet included are called unsatisfied references. You can manually force additional blocks into the output file by using the ADD command.

Although input files can include several copies or versions of the same routine, Binder includes only one copy in the new object code file. The copy that is chosen depends on the ordering of Binder commands and on the order in which Binder explores the search list.

The Input Control Lists

During the input stage, Binder accepts commands and collects the information in control lists. These lists are used to build the target file during the output stage. The input control lists follow:

- Include lists, specifically:
    - Include code block list

    - Include entry point list
    - Include data block list
    - Include run-time data unit list (RTDU list)
- Modify list
- Omit list
- Refer list
- Search list
- Undefined list
- Unresolved reference lists, specifically:
    - Unresolved code list
    - Unresolved data list

When you start Binder, all lists are empty. After Binder builds the target file, the lists are again empty. You can then define another target file or end the session.

Creating the Input Control Lists

You create the input control lists with Binder commands entered during the input stage. Binder automatically maintains the unresolved reference lists. You do not enter commands to create these lists.

Table 5-1. Commands That Create Control Lists

Command	Description
ADD	Places names on the include lists.
REPLACE	Changes code blocks, their associated entry points and RTDUs, and data blocks on the appropriate include lists.
SELECT	Adds names to the omit, search, and refer lists.
MOVE	Reorders the include entry name list and include code list.
SATISFY	Resolves entry point names and RTDUs on the unresolved entry point list and data block names on the unresolved and undefined data lists.
CLEAR	Clears (resets to empty) the include, omit, search, refer, unresolved reference, undefined data, and modify lists.
MODIFY	Creates the modify list by specifying a set of modifications when building the target file.

How Binder Uses the Input Control Lists

The following subsections describe the input control lists and how and when Binder adds and removes blocks and entry points from the lists.

Include Lists

The include lists are ordered lists of code blocks, data blocks, entry points, and RTDUs to include in the target file. The ADD or REPLACE commands add the names of code blocks, data blocks, entry points, and RTDUs to the applicable include list in the order specified. Binder adds names from subsequent ADD commands to the end of the list. REPLACE puts the replaced blocks in the same place on the list.

ADD and REPLACE commands can refer to entry points explicitly by name or implicitly as part of a range of entry point names. Inserting a range of entry points specifies that entry points in the file (in physical order) from the beginning to the end of the range are to be added to the include list. All primary entry point names within the range are added to both the include code block list and the include entry point list; secondary entry point names are only added to the include entry point list. Names of own data blocks for the code blocks are added to the include data block list.

If a code block added to the include code block list contains a reference to an entry point, data block, or RTDU that is not in the respective include list, Binder adds the name to the unresolved reference list. Binder tries to resolve the reference when you execute a SATISFY or BUILD command. (You can prevent automatic resolution of entry points and RTDUs by using the SELECT SATISFY OFF command.)

Include Code Block List.

Binder adds a code block name to the include code block list when one of the following happens:

- An ADD or REPLACE command refers to the code block (primary entry point) name explicitly or implicitly as part of a range of entry points.
- An ADD or REPLACE command refers to the total contents of a disk file containing the code block.
- Binder executes a SATISFY or BUILD command and the following are true:
  - An entry point, referenced by an included code block, is in the unresolved reference list.
  - The entry point name is not in the omit list.
  - The code block containing the entry point that can satisfy the reference is in a disk file in the search list.

The order of code block names in the include code block list generally determines the order in which code blocks are allocated during the output phase. Binder might change the order of code block names in three cases:

- When a code block is greater than 32K words
- When a code block is a resident procedure
- When a gap in a code segment could be filled by a small code block

Using the MOVE command, you can reorder the include code block list during the input phase in such a way that page faults are avoided during execution. For diagnostic information on using MOVE in this way, see the *Measure Reference Manual*

Include Data Block List.

Binder adds a data block name to the include data block list if any one of a set of conditions is true:

- The data block is a common block, and an ADD or REPLACE command refers to the data block name either explicitly or implicitly as part of a range of data blocks; ordering is not necessary.
- The data block is a common block referenced by a code block in the include code block list.
- The data block is the own (private) block of a code block in the include code block list.
- The data block is part of a disk file whose total contents are referred to by an ADD or REPLACE command.
- The data block is a special data block required by a code block in the include code block list.
- Binder executes a SATISFY or BUILD command and the data block, referenced by an included code block, is in the unresolved reference list.

Include Entry Point List.

Binder adds an entry point name to the include entry point list if any of the following is true:

- An ADD or REPLACE command refers to the entry point name explicitly or implicitly as part of a range of entry points.
- An ADD or REPLACE command refers to the total contents of a disk file containing the entry point. Secondary entry points, as well as the primary entry points, are added to the list.
- The code block including the secondary entry point is put on the include list.

The order of entry point names in this list corresponds to the order of code block names in the include code block list. Binder positions an entry point name in the same order as that of the ADD or REPLACE command that refers to it.

Include Run-Time Data Unit List.

Binder adds an RTDU name to the include run-time data unit list when one of the following happens:

- An ADD or REPLACE command refers to the total contents of a disk file containing the RTDU.
- Binder executes a SATISFY or BUILD command and the following are true:
  - An RTDU, referenced by an included code block, is in the unresolved reference list.
  - The code block containing the RTDU that can satisfy the reference is in a disk file in the search list.

Modify List

You can change the contents of an existing code block or data block when Binder builds the target file. During the input stage, you specify modifications using the MODIFY command. Binder saves the set of modifications established by this command in the modify list and makes the actual changes when it builds the target file (after fixups). You cannot modify uninitialized data blocks.

Omit List

The omit list contains the names of entry points that Binder must exclude from the target file even if the included code refers to these entry points. Likewise, it does not matter whether the files on the search list contain the entry points. You can use the omit list to force an entry point to be satisfied at run time, either from a system library or from a user library.

You cannot specify an entry point name for the omit list if the name is already in the include entry point list.

The SELECT OMIT command specifies entries for the omit list. You can override the omit list by specifying a different omit list in the SATISFY or BUILD command. The SATISFY or BUILD respecification is in effect only during execution of these commands.

Note that giving the BUILD command implies a SATISFY command unless you specify SELECT SATISFY OFF, which prevents automatic resolution of entry point references. The SATISFY command refers to entry points and data blocks; SELECT SATISFY refers only to entry points.

Clear the current omit list by using the RESELECT OMIT command.

Refer List

The refer list is a list of pairs of entry point names. The first entry point name of the pair is the existing name used to refer to an entry point (the old name). The second entry

point name of the pair is the name of an entry point (the new name) that is to be substituted for the existing entry point name. The two names cannot be the same.

When Binder executes a SATISFY or BUILD command, it checks unresolved entry point references against the refer list. If the reference is to be changed, Binder makes the change and then tries to resolve the reference. The refer list does not apply to entry points previously added to the include entry point list.

Binder builds the refer list according to the specifications you give using the REFER parameter of the SELECT command. If you specify the REFER parameter in the SATISFY or BUILD command, the refer list of the SATISFY or BUILD command takes precedence over the refer list of the SELECT command.

You can clear the refer list by using the REFER parameter of the RESELECT command.

Search List

The search list contains the disk file names of object files to search in order to resolve entry point and data block references. Binder uses this list when executing a SATISFY or BUILD command. Binder searches the files in the order in which the file names are listed for blocks that define entry points and data blocks in the unresolved and undefined reference lists.

Binder builds the search list according to the specifications you give for the SEARCH parameter of the SELECT command. If you specify the SEARCH parameter in the BUILD or SATISFY command, the search list of the SATISFY or BUILD command takes precedence over the search list of the SELECT command.

Undefined List

Binder creates an entry in the undefined list for any data block references that are not initialized in an object file. This applies to C and Pascal binding only. Note that the object file is not runnable until these blocks are initialized.

Unresolved Reference Lists

The unresolved reference lists contain entry point, data block, and RTDU names that are referred to but are not defined by any block in the include lists. Binder notes whether the reference is internal or external to the file that contains the reference at the time it puts the reference on the unresolved reference list. The when Binder executes a SATISFY or BUILD command, it tries to resolve each entry point, data block, or RTDU name as described below.

For entry points, Binder performs the following steps:

1. Binder searches the refer list for a redirection of the entry point.

2. Binder searches the omit list; if it finds the name in the omit list, it does not resolve the entry point name. No further action takes place for that reference and the entry point name remains in the unresolved reference list.
3. If the reference is internal to the file that contained it, Binder searches that file. If the reference is external to the file that contained it, Binder searches the object files named in the search list for an entry point that satisfies the reference; Binder searches object files in the order of the file names in the search list. If Binder finds the entry point, it adds the code block that defines the entry point to the end of the include code block list and deletes the entry point name from the unresolved reference list.
4. If adding a code block to the include code block list introduces further unresolved references, Binder checks the refer list for redirection of those entry points and notes whether the reference came from the same file as the block that was just included. If an entry point is redirected and can be resolved by the include entry point list, no further action takes place; otherwise, Binder adds the entry point name to the unresolved reference list.

For data block names, Binder tries to resolve each reference by searching the file containing the code block that referenced the common block. Binder performs this step regardless of the setting of SELECT SATISFY command.

For RTDUs, if the reference is internal to the file that contained it, Binder searches that file. If the reference is external to the file that contained it, Binder searches the object files named in the search list for an RTDU that satisfies the reference; Binder searches object files in the order of the file names in the search list. If Binder finds the RTDU, it adds the code block that defines the RTDU to the end of the include code block list and deletes the RTDU name from the unresolved reference list.

Once Binder has built the target file, entry points, RTDUs, and data blocks are handled differently. For entry points and RTDUs, any unresolved reference becomes an external reference that must be satisfied either by a subsequent Binder operation or at run time. For the languages currently supported by Binder, no data block names should appear in the unresolved reference list after binding.

The following example illustrates how Binder resolves unsatisfied references when building the target file. This example assumes that:

- The unresolved entry list specifies A, B, C, and D (in that order).
- Procedure A calls procedure B and procedure B calls procedures C and D.
- File OBJ1 contains procedures A and C and file OBJ2 contains procedures B, C, and D.

The following commands produce the results specified in the subsequent paragraphs.

```
@ADD CODE A FROM OBJ1

@SELECT SEARCH OBJ1

@SELECT SEARCH OBJ2
```

```
@SATISFY
```

Binder resolves the reference to A from OBJ1, and then resolves the reference to B from OBJ2. Because B calls C and D, Binder adds the references to C and D to the unresolved entry list. It also notes that C and D are contained in the same file as B. When it comes time to resolve the reference to C, Binder does so by searching file OBJ2. Finally, Binder resolves D from OBJ2.

If A had also called C, Binder would have added C to the unresolved entry list before resolving B and would have noted that C was contained in OBJ1. In that case, C would have been resolved from OBJ1.

It is important to be aware of the order in which calls are resolved, because two procedures having the same name do not necessarily contain the same code. This can lead to unexpected results.

The Target File

The output stage begins when the BUILD command causes an implicit satisfy for unresolved references. During the output stage, Binder builds the target file according to the names in the include lists, writes out listings, and clears all of the internal lists (include, omit, refer, search, unresolved reference, undefined, and modify).

Target File Attributes

Binder builds the target file using the attributes specified with the SET command or by the set-param parameter of the BUILD command.

Table 5-2. Target File Attributes (page 1 of 3)

Attribute	Description
DATA	Specifies the total amount of nonextended data pages allocated at run time for data blocks, stack, and local storage. The default is 64K words for C and Pascal. Note that you can use only one of DATA, STACK, or EXTENDSTACK to override the default amount of data space allocated by Binder.
EXTENDSTACK	Specifies an amount of stack space to add to the amount of stack space estimated by Binder. Binder then allocates the total amount of stack space and total data block space. Note that you can use only one of DATA, STACK, or EXTENDSTACK to override the default amount of data space allocated by Binder. The default data space allocated by Binder is the amount of space required for all of the data blocks plus an estimated amount of stack space for local storage.
HEAP	Sets the maximum size of the heap used in C and Pascal. The value you specify (in words, pages, or bytes) overrides any other value Binder has encountered. The default is 0.
HIGHPIN	Specifies that the object file can run at a high process identification number (PIN) if one is available. The default is OFF.
HIGHREQUESTERS	Specifies that the object file can support requests from processes running at a high process identification number (PIN). The default is OFF.
INSPECT	Specifies which debugging program (Inspect or Debug) you want to use when the object code is executed. The default is OFF; that is, the Debug program is used.
LARGESTACK	Sets the size of the $EXTENDED#STACK data block used in TAL. The value you specify overrides any other value Binder has encountered. If you omit this entry, Binder uses the largest $EXTENDED#STACK block possible.
LIBRARY	Specifies the user library to associate with the object file at run time. If you omit this entry, Binder associates no user library with the object file. You can override this file attribute at run time by specifying the LIB parameter in the command interpreter RUN command.
LIKE	Specifies that the following attributes of the target file are to be the same as those of a specified object file: DATA, INSPECT, LIBRARY, and SAVEABEND.

Table 5-2. Target File Attributes (page 2 of 3)

Attribute	Description
PEP	Explicitly specifies a larger size for the PEP table. By default, Binder allocates the minimum amount of space needed for the number of entry points in the object file.
PFS	Specifies the size of the Process File Segment. The PFS can be from 64 to 512 pages.
RUNNAMED	Specifies that the object file must be run as a named processes. The operating system assigns a name if a name is not specified at process creation. The default is OFF.
SAVEABEND	Determines whether the Inspect symbolic debugger creates a save file if the process terminates abnormally during execution. Binder verifies that Inspect is ON if SAVEABEND is ON. The SAVEABEND default is OFF. The operating system creates the save file in the same volume and subvolume as the program and assigns it a name in the format of ZZSA nnnn. This file contains information on the process environment at the point of termination including: ● Names of all open files ● A copy of the data space at the time the process terminated ● Name of the process and a timestamp for the time of termination
STACK	Specifies the amount of stack space to allocate for local storage. Binder adds this parameter value to the amount of space required for all data blocks, then allocates the total amount of data space. Note that you can use only one of DATA, STACK, or EXTENDSTACK to override the default amount of data space allocated by Binder.
SUBTYPE	Sets the value of process subtype for the target file. The default value is 0.
SYMBOLS	Specifies whether Binder retains the symbol tables of blocks on the include lists, in the target file’s Inspect region. (When you use the SYMBOLS compiler directive, Binder includes a symbol table in the object file.) SYMBOLS ON is the default and specifies that Binder retains the tables in the object file.

Table 5-2. Target File Attributes (page 3 of 3)

Attribute	Description
SYSTYPE	Specifies whether the target execution environment for the object file is Guardian environments or OSS environment. The default is Guardian.
TARGET	Specifies the processor that the target file can run on. TARGET can be TNS, TNS/R, ANY, or unspecified. The default is unspecified.
USERLIBRARY	Specifies whether Binder builds a target file with a maximum of 16 or 32 code segments. If you specify the latter, the process creation system procedure call creates a process with the first 16 code segments in user code space and the remaining segments in the user library space. The default value, OFF, specifies a maximum of 16 code segments.

Note that the Inspect symbolic debugger also provides an interactive method for saving the environment of a process as well as examining save files. See the descriptions of the SAVE and PR commands in the *Inspect Manual*.

How Binder Builds the Target File

When the BUILD command executes, Binder creates the target file according to the code block, data block, entry point, and RTDU names currently on the include lists. Binder performs the following steps in creating the target file:

1. It tries to satisfy entry point, data block, and RTDU references in the unresolved and undefined reference lists.
2. It defines entry points that cannot be resolved as external references for the object file.
3. It allocates any uninitialized common block in an area preset to zero.
4. It copies blocks named in the include lists to the target file.

Binder allocates code blocks contiguously in the order in which the code block names appear in the include code block list. The RESIDENT attribute overrides the include list order.

If you have set the COMPACT parameter of the SELECT command to ON (the default), Binder checks each succeeding code block to determine whether the block fits in a gap between the previous code block and the 32K boundary. If Binder finds such a block, Binder allocates the block in that gap. Use of the SELECT COMPACT ON command can result in nonresident code blocks preceding resident code blocks.

The CHECK parameter of the SELECT command can also affect the construction of the target file. This parameter selects the different types of error checking that Binder provides during the binding operation. The available options are as follows:

- The BLOCK option causes common block declarations to be checked for the same length and the same type of addressing in every code block that refers to the common block. If the length and type do not match for each of the references, Binder issues a warning. The default is ON.
- The LIBRARY option causes the following checking to occur when Binder builds a user library:
  - Binder checks code blocks being placed into the object file for references to data blocks other than read-only blocks.
  - Binder also checks code blocks for the MAIN attribute.

  If it finds either, Binder issues a warning message. The default is OFF.
- The PARAMETER option specifies the extent to which Binder checks parameter lists, function return values, and language consistency.

  Binder provides five levels of parameter, return value, and language checking: ON, STRICT, STRONG, LENIENT, and OFF. The default setting is STRICT.

Depending upon the setting of the PARAMETER option, Binder checks the parameter lists for the respective code blocks for consistency in size, type, and mode. Mode checking refers to whether the parameter is passed by value or by reference. Binder issues a warning message if the called code block's parameter requirements do not match those of the caller. Binder checks to make sure that the caller specifies the correct language for the called routine; it issues a warning message if the caller specifies a different language than that of the called routine. If the caller explicitly states that the language of the invoked routine is unspecified, Binder does not perform this check. Refer to the discussion in SELECT Command on page 3-49 for more information.

Note. You might receive parameter mismatch errors when using BIND that you did not receive when using BINSERV. This can occur if you compile with parameter checking turned off and then start a BIND session with the parameter checking turned on. In particular, the parameter mismatch messages are likely to occur in mixed-language binding.

---

6 User Libraries

This section contains the following topics:

Topic	Page
Binding User-Library Procedures	6-1
Object File Format	6-2
Preventing Binder Resolution of Library Calls	6-2
Specifying a User Library	6-3
Restrictions on User Libraries	6-3
Shared Run-Time Libraries	6-4

A user library is a set of procedures that the operating system can link to a program file at run time. This section contains general information needed for effective programming with user libraries.

User libraries are available in TAL and FORTRAN and D-series C and COBOL85 programs. User libraries are not available in Pascal.

User libraries provide many benefits to programs. You can place commonly used procedures in a user library:

- To reduce the storage required for object code on disk and in main memory.
- To share a set of common procedures among applications.
- To extend a single application's code space.

Binding User-Library Procedures

The first time you execute a program file after compilation, the system searches the optional user library to resolve each unresolved external reference before searching the system code and library.

The HP NonStop Operating System resolves an external reference by changing the call in the program file appropriately; that is, to point to the user library or to the system library. You can then run the program file repeatedly without satisfying the references again.

If the operating system cannot find a user or system library procedure to satisfy a run-time external reference, it displays a message as the process starts. When the process makes a call to an unresolved procedure, the process changes the reference into a call to the Debug utility, and the process enters the debug state.

Run-time binding does not include copying the procedure into the program file. A program file can have only one user library associated with it. Therefore, on a TNS system, a running program can have 16 segments of program code space and 16 segments of library code space for a total of 32 segments. On a TNS/R system, a running program can have 32 segments of program code space and 32 segments of

library code space for a total of 64 segments. A segment can contain as many as 64K words of code. For instructions about using the USERLIBRARY option to increase the size of a program's code space from 16 to 32 segments (for TNS systems) and from 32 to 64 segments (for TNS/R systems), see SET Command on page 3-57.

Object File Format

Because the operating system binds program files and library procedures at execution time, there are no restrictions on object-file format. A program file and its library file can exist in any combination of new Binder and old system formats.

Preventing Binder Resolution of Library Calls

If you plan to use a user library (for example, to avoid code-space overflow), make sure Binder does not insert library procedures into the object file before run time. This can be accomplished at compile time or during a command-driven BIND session.

Compilation-Time Binding

At compilation time, Binder tries to resolve external references if the compiler SEARCH directive or pragma specifies a list of object files for this purpose. The exception to this is COBOL85's use of the file C8LIB. For more information, see the *COBOL85 Reference Manual*.

When a search list is present, the Binder tries to resolve all unresolved external references. If compilation-time binding places a user-library procedure in the program file, you can delete that procedure from the program file in an independent Binder session.

Command-Driven Binding

During a Binder session, you can specify names of individual entry points that Binder should not include in an object file. First build a user-library file of procedures copied from existing program files; then, build more compact program files by omitting the library procedures.

To specify the external references that Binder should not resolve, use these commands as appropriate:

- The SELECT OMIT command to entry points of user library procedures.

  This prevents binding of these procedures in the program file.

- The BUILD command with the SATISFY OFF parameter option.

  This, however, prevents any attempt at resolution after the target file contents have been specified by other commands.

- The DELETE command to remove user-library procedures that were already bound in the program file.

Specifying a User Library

Only one user library can be associated with a program file at any time. Therefore, all concurrently executing processes created from a single program file use the same library file.

You specify the user library for a program file by using one of the following:

- The COBOL85, FORTRAN, and TAL compiler directive LIBRARY
- The Binder command SET LIBRARY
- The command interpreter RUN command LIB option

The Binder SET LIBRARY command overrides any LIBRARY compiler directives. The RUN command LIB option overrides both the Binder SET LIBRARY commands and the LIBRARY compiler directives.

The library remains associated with the program file until you explicitly change it. That is, the operating system changes to the program file for external-reference resolution remain when the process completes. If the operating system detects modifications to the program file or the user library when either file is specified in a subsequent RUN command, the operating system again resolves the calls left for resolution.

See the BUILD Command on page 3-11, CHANGE Command on page 3-15, and SET Command on page 3-57 to learn how to specify and change user libraries.

To specify a different library with a program file, you need write access to the program file, and all processes created from the program file must have stopped.

Restrictions on User Libraries

The SELECT CHECK LIBRARY ON command directs Binder to enforces rules to which user libraries must adhere. Specify this command when building a target file that will be used as a user library. Binder issues a warning for any reference to a data block other than a read-only block or any entry point that has the MAIN attribute. Note that BIND does not check existing libraries, only user libraries that are being built.

User library files must meet the following restrictions:

- A library file cannot contain a main program. If it does, a run-time error results.
- Library routines cannot call routines in the program file.
- FORTRAN user libraries cannot contain DATA statements, SAVE statements, or COMMON statements.
- C user library routines can be called by routines written in any HP programming language, but the calling program file must contain at least one C routine. This is required to ensure the C run-time library resources are available to the user library.

Additionally, user library files must meet the following restrictions on data declarations:

- There is one data space (stack) for a process, the one allocated for the program file.
- The library file's global data matches the global data of the program file. Because you specify the allocation and initialization of global data in your program file, if the library file has global data, it must match the global data of the program file. The operating system ignores any initialization of global data in the library file.
- The library file cannot contain global data if the program file contains embedded SQL statements. These global data blocks are used to process the embedded SQL statements at run-time.
- A TAL library procedure can have its own read-only arrays. Nevertheless, a global read-only array must be in the code space containing references to the array. If both code spaces contain references to such an array, copies of the array must exist in both the library and in the program file.
- User-library procedures cannot pass read-only array arguments to user code or to system code. In particular, a user-library call to the Guardian system call procedure FORMATDATA cannot pass read-only array arguments. For language specific information on user libraries, see the appropriate manuals for more information.

Shared Run-Time Libraries

Shared run-time libraries (SRLs) are special user libraries that contain global variables. The SRLs are provided by HP. Building an application that uses an SRL is similar to building an application that uses a user library. This section describes some additional commands used with SRLs.

Note. A program file can use either a user library or an SRL.

Building Applications That Use SRLs

When building an application, you must specify an existing library to import global variables from and reserve space in primary, secondary, and extended memory to store the library variables. Binder uses the recommended data space sizes from the library, but you can adjust these space sizes. Variables explicitly exported by the library can be accessed in the application by name.

Using Binder Commands With SRLs

The following Binder commands support shared run-time libraries:

Table 6-1. Binder Commands Used With Shared Run-Time Libraries

Command	Description
SELECT IMPORT LIBRARY	Specifies the use of an SRL library as a user library used during binding.
SET IMPORT	Specifies the list of variables to be imported.
SET RESERVE	Specifies how much space to reserve in the application's global data space.
SHOW IMPORT	Displays the list of imported data blocks
SHOW RESERVE	Displays the reserve settings for each of the four data areas.

See Section 3, BIND Commands for a complete description of these commands.

Reserving Space With the SET RESERVE Command

The following considerations apply when reserving space in applications:

- The application reserves as much BELOW64, PRIMARY, SECONDARY, and EXTENDED data space as is recommended in the library. Binder reads this information from the user library object file header when the SELECT IMPORT LIBRARY command is specified.
- The application builder can change the size of each reserved area, whether by increasing it, by decreasing it, or by setting it to an absolute amount. The incremental values are combined for each data area, to determine the final size of space to reserve.
- If an absolute value is specified, it overrides any previous absolute or incremental setting for that data area.
- If the final size is smaller than required for the current instance of the library, Binder reports a warning. The application is built even though the reserved size is insufficient.
- A new library can be supplied which uses less space, but most likely the application needs to be rebound using a larger reserve space.
- If the application does not reserve enough space, the system gives a fatal error at process initiation time.

7 Guardian File Names and TACL Commands

This section discusses:

Topic	Page
Disk File Names	7-1
TACL Commands	7-4
TACL DEFINE Commands	7-5
TACL PARAM Commands	7-7
TACL ASSIGN Commands	7-9

The Binder product assumes that file names supplied for input and output follow NonStop Operating System naming conventions. Defaults are supplied by the command interpreter when Binder is started.

For information on process or device file names, see the *Guardian Programmer's Guide*.

Disk File Names

A disk file name identifies a file that contains data or a program. A disk file name reflects the specified file's location on an HP system. The location of a disk file on an HP system is analogous to the location of a form in a file cabinet. To find the form, you must know:

- Which file cabinet it is in
- Which drawer it is in
- Which folder it is in
- Which form it is

Analogously, to find a disk file on an HP system, you must know:

- Which node (system) it is on
- Which volume it is on
- Which subvolume it is on
- Which disk file it is

In general, disk file names:

- Cannot contain spaces

- Can contain ASCII characters only
- Are not case-sensitive; the following names are equivalent:

  ```
  myfile
  MyFile
  MYFILE
  ```

  Language functions and system procedures that return file names might return them in uppercase (even if the file name was originally in lowercase). Check the description of the function or procedure that you are using.

Parts of a Disk File Name

A disk file has a unique file name that consists of four parts, with each part separated by a period:

- A node name
- A volume name
- A subvolume name
- A file ID

Here is an example of a disk file name:

```
\mynode.$myvol.mysubvol.myfile
```

You can name your own subvolumes and file IDs, but nodes (systems) and volumes are named by the system manager.

All parts of the file name except the file ID are optional except as noted in the following discussion. If you omit any part of the file name, the system uses values as described in Partial File Names on page 7-3, later in this section.

Node or System Name

The node or system name, such as \MYNODE, is the name of the node or system where the file resides. If specified, the node or system name must begin with a backslash (\) followed by one to seven alphanumeric characters. The character following the backslash must be an alphabetic character.

Volume Name

The volume name, such as $MYVOL, is the name of the disk volume where the file resides. If specified, the volume name must begin with a dollar sign ($), followed by one to six or one to seven alphanumeric characters as follows. The character following the dollar sign must be an alphabetic character.

The volume name can contain one to seven alphanumeric characters.

Subvolume Name

The subvolume name, such as MYSUBVOL, is the name of the set of files, on the disk volume, within which the file resides. The subvolume name can contain from one to eight alphanumeric characters, the first of which must be alphabetic.

On a D-series system, if you specify the volume name, you must also specify the subvolume name. If you omit the volume name, specifying the subvolume name is optional.

File ID

The file ID, such as MYFILE, is the identifier of the file in the subvolume. The file ID can contain from one to eight alphanumeric characters, the first of which must be alphabetic.

The file ID is required.

Partial File Names

A partial file name contains at least the file ID, but does not contain all the file-name parts. When you specify a partial file name, the operating system or other process fills in the missing file-name parts by using your current default values. Following are the optional file-name parts and their default values:

File-Name Part	Default
node (system)	Node (system) on which your program is executing
volume	Current default volume
subvolume	Current default subvolume

Following are all the partial file names you can specify for a disk file named \BRANCH.$DIV.DEPT.EMP:

Omitted File-Name Parts	Partial File Name	Permitted on D-Series and G-series Systems
Node (system)	$div.dept.emp	Yes
Node (system), volume	dept.emp	Yes
Node (system), volume, subvolume	emp	Yes
Volume	\branch.dept.emp	Yes

Omitted File-Name Parts	Partial File Name	Permitted on D-Series and G-series Systems
Volume, subvolume	\branch.emp	Yes
Subvolume	\branch.$div.emp	No
Node (system), subvolume	$div.emp	No

You can change your current default values in various ways:

- You can change the volume and subvolume with the VOLUME command of, for example, the Binder, Inspect, and TACL products.
- In some cases, you can specify node (system), volume, and subvolume names by issuing TACL ASSIGN SSV commands, described later in this section.

Logical File Names

You can use a logical file name in place of the disk file name. A logical file name is an alternate name you specify in a TACL DEFINE or TACL ASSIGN command, described later in this section.

Internal File Names

The C-series operating system uses the internal form of a file name when passing it between your program and the operating system. The D-series operating system uses the internal form only if your program has not been converted to use D-series features.

For information on converting external file names to internal file names in a program, see the *Guardian System Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

TACL Commands

You can send information to the compiler by using the following TACL commands:

- DEFINE
- PARAM
- ASSIGN

These commands are summarized in the remainder of this section. For complete information on these commands, see the following manuals:

- *TACL Reference Manual* (syntactic information)
- *TACL Programmer's Guide* (programmatic information)

- *Guardian User's Guide* (interactive information)
- *Guardian Programmer's Guide* (programmatic information)

TACL DEFINE Commands

By issuing TACL DEFINE commands before starting the compiler, you can:

- Substitute a file name for a DEFINE name used in the source file
- Specify spooler attributes
- Specify file attributes on a labeled tape
- Specify process defaults, such as default volume and subvolume

Substituting a File Name

You can substitute a file name for a DEFINE name being passed by a nonprivileged program to a system procedure. To substitute a file name, issue the following TACL commands:

TACL Command	Purpose
SET DEFMODE ON	Enable DEFINE processing
SET DEFINE CLASS	Set the initial attribute of a DEFINE command to CLASS MAP
SET DEFINE	Set the working attributes
ADD DEFINE	Specify a file name to substitute for a DEFINE name

TACL DEFINE Names

TACL DEFINE names:

- Are not case-sensitive
- Consist of 2 to 24 characters
- Begin with an equals sign (=) followed by an alphabetic character
- Continue with any combination of letters, digits, hyphens (-), underscores (_), and circumflexes (^)

Some examples of valid DEFINE names are:

```
=A

=The_chosen_file

=Long-but-not-too-long

=The-File-of-The-Week
```

DEFINE names that begin with an equals sign followed by an underscore (=_) are reserved by HP. For example, do not use DEFINE names such as =_DEFAULT.

Setting DEFINE CLASS Attributes

To create a DEFINE message or set its attributes, you must set a CLASS attribute for the DEFINE. The CLASS attributes are MAP, TAPE, SORT/SUBSORT, SPOOL, and DEFAULTS. Each attribute has an initial setting based on whether the attribute is required, optional, or default.

MAP DEFINE

When you log on, the default CLASS attribute is MAP, which requires a file name. A MAP DEFINE specifies a substitute file name for a file name. For example, suppose that your current CLASS attribute is MAP and you have created a DEFINE called =MYHOME for the file \SANTA.$CLARA.CALI.FORNIA.

To add all code and data blocks from \SANTA.$CLARA.CALI.FORNIA, you would enter:

```
15> BIND

@ADD * FROM =MYHOME
```

You can use the TACL INFO command to find information about a DEFINE:

```
14> INFO DEFINE =MYHOME

    Define Name =MYHOME

    CLASS       MAP

    FILE        \SANTA.$CLARA.CALI.FORNIA
```

TAPE DEFINE (D-Series Systems Only)

The TAPE DEFINE lets you specify attributes for labeled magnetic tapes. For instance, it lets you specify attributes such as block length, recording density, record format and length, number of reels, and labeling.

SPOOL DEFINE

The SPOOL DEFINE lets you specify attributes such as number of copies, form name, location, owner, report name, and priority.

DEFAULTS DEFINE

In the DEFAULTS class, a permanently built-in DEFINE named =_DEFAULTS has the following attributes, which are active regardless of any DEFMODE setting:

Attribute	Required	Purpose
VOLUME	Yes	Contains the default node, volume, and subvolume names for the current process as set by the TACL VOLUME, SYSTEM, and LOGON commands
SWAP	No	Contains the node and volume name in which the operating system is to store swap files
CATALOG	No	Contains a substitute name for a catalog as described in the *NonStop SQL Reference Manual*

TACL PARAM Commands

The compilers accept TACL PARAM commands that you issue before starting the compilers. PARAM commands are BINSERV, SAMECPU, SWAPVOL, and SYMSERV.

PARAM BINSERV Command

The PARAM BINSERV command lets you specify which BINSERV process you want to use. You can specify a file name or a TACL DEFINE name.

For example, you can specify the BINSERV file on a particular node, volume, and subvolume as follows:

```
PARAM BINSERV \mynode.$myvol.mysubvol.BINSERV
```

If the specified node is not the one the compiler runs on, the compiler ignores the command. If you omit the volume and subvolume, the compiler uses the current default volume and subvolume. If you omit the file ID, the compiler uses the file ID BINSERV. If you specify a TACL DEFINE name, it must refer to a disk file of class MAP.

If you use this command, the error file PDTERROR must be on the same subvolume as BINSERV. If you omit this command, the compiler uses the BINSERV process on its own volume and subvolume.

PARAM SAMECPU Command

The PARAM SAMECPU command causes the compiler, BINSERV, and SYMSERV to all to run in the same CPU if you specify any number but 0. For example:

```
PARAM SAMECPU 32767
TAL /CPU 6/
```

Specifying 0 means the compiler, BINSERV, and SYMSERV need not run on the same CPU. For example:

```
PARAM SAMECPU 0
```

PARAM SWAPVOL Command

The PARAM SWAPVOL command lets you specify the volume that the compiler, BINSERV, and SYMSERV use for temporary files. For example:

```
PARAM SWAPVOL $myvol
```

The compiler ignores any node specification and allocates temporary files on its own node. If you omit the volume, the compiler uses the default volume for temporary files; BINSERV and SYMSERV use the volume that is to receive the object file.

Use this command when:

- The volumes normally used for temporary files might not have sufficient space.
- The default volume or the volume to receive the object file is on a different node from the compiler.

Note that the C and Pascal compilers ignore PARAM SWAPVOL commands. The SWAP volume option of the Pascal and C compilation commands determines the swap volume used by the BINSERV process. If you do not specify the SWAP volume option in a Pascal or C compilation command, the BINSERV process uses the volume on which the compiler code file resides as the swap volume. Compiler code usually resides on volume $SYSTEM.

PARAM SYMSERV Command

The PARAM SYMSERV command lets you specify which SYMSERV process you want to use. You can specify a file name or a TACL DEFINE name.

For example, to specify the SYMSERV file on a particular volume and subvolume:

```
PARAM SYMSERV \mynode.$myvol.mysubvol.SYMSERV
```

If the node is not the one the compiler runs on, the compiler ignores the command. If you omit the volume or subvolume, the compiler uses the current default volume or subvolume. If you omit the file name, the compiler uses the name SYMSERV. If you specify a TACL DEFINE name, the name must refer to a disk file of class MAP.

If you omit this command, the compiler uses the volume and subvolume specified in the PARAM BINSERV command. If you omit both PARAM SYMSERV and PARAM BINSERV commands, the compiler uses the SYMSERV process on its own volume and subvolume.

Using PARAM Commands

You can specify one or more PARAM commands before starting the compiler. For example, you can specify that:

- The compiler use the BINSERV process located on MYSUBVOL

  ```
  PARAM BINSERV mysubvol
  ```

- The compiler, BINSERV, and SYMSERV all run in the same CPU

```
PARAM SAMECPU 1
```

- The compiler, BINSERV, and SYMSERV allocate temporary files on volume $JUNK

```
PARAM SWAPVOL $junk
```

Then you can issue the compiler compilation command:

```
TAL /IN mysource, OUT mylist/ myprog
```

TACL ASSIGN Commands

You can issue the TACL ASSIGN command before starting the compiler to substitute file names for those used in the source file. The TACL product stores the file-name mapping until the compiler requests it.

The ASSIGN command equates a file name with a logical file name used in ERRORFILE, SAVEGLOBALS, SEARCH, SOURCE, and USEGLOBALS directives. The compiler accepts only the first 75 ordinary ASSIGN messages.

In each ASSIGN command, specify a logical identifier followed by a comma and the file name or a TACL DEFINE name:

```
ASSIGN dog, \a.$b.c.dog

ASSIGN cat, =mycat
```

If the file name is incomplete, the TACL product completes it from your current default node, volume, and subvolume. For example, if your current defaults are \X.$Y.Z, the TACL product completes the incomplete file names in ASSIGN commands as follows:

Incomplete File Names	Complete File Names
ASSIGN qq, cat	ASSIGN qq, \x.$y.z.cat
ASSIGN ss, b.dog	ASSIGN ss, \x.$y.b.dog
ASSIGN tt, $a.b.rat	ASSIGN tt, \x.$a.b.rat.

If you use a TACL DEFINE name in place of a file name, the TACL product qualifies the file name specified in the ADD DEFINE command when it processes the ASSIGN command. Even if you specify new node, volume, and subvolume defaults between the ADD DEFINE command and the ASSIGN command, the ASSIGN mapping still reflects the ADD DEFINE settings.

If you issue the following commands:

```
ASSIGN aa, $a.b.cat

ASSIGN bb, $a.b.dog

ASSIGN cc, =my_zebra
```

```
ADD DEFINE =my_zebra, CLASS MAP, FILE $a.b.zebra

TAL /IN mysource, OUT $s/ obj
```

the compiler equates SOURCE directives in MYSOURCE to files as follows:

```
?SOURCE aa        !Equates to ?SOURCE $a.b.cat

?SOURCE cc        !Equates to ?SOURCE $a.b.zebra

?SOURCE bb        !Equates to ?SOURCE $a.b.dog
```

You can name new source files at each compilation without changing the contents of the source file.

8 Binder Messages

This section lists three types of messages returned by Binder: error messages, warnings, and completion codes.

Topic	Page
Error Messages and Warnings	8-1
Completion Codes	8-28

Error Messages and Warnings

This subsection lists Binder error and warning messages in numerical order. The listing includes a description of the possible sources of the error and the corrective action you can take.

The Program Development Tools (PDT)—that is, Binder, Crossref, and Inspect—all have the error file called PDTERROR associated with them. This file must reside on the same volume and subvolume as the PDT products. As part of the standard RVU, they reside on $SYSTEM.SYSTEM. If you choose to relocate the PDT products on some other volume or subvolume, you must also relocate PDTERROR and PDTHELP. When Binder detects an error condition, it searches PDTERROR for the corresponding error message (documented in this section). If PDTERROR is missing, Binder displays a cryptic message of the form

```
BINDER ERROR nn
```

and warns you that the error file is missing.

All Binder error messages are qualified by a letter representing their severity code. These codes and their meanings are as follows:

Code	Meaning
W	Warning. Binder issues a WARNING but still produces an object file.
E	Error. Binder returns to the Binder prompt without completing the current operation.
F	Fatal Error. Binder terminates without producing an object file.

```
F Internal error at P=% nnnnnn text
```

This message precedes other information about a Binder internal consistency error; a stack trace usually accompanies the message. If you receive this message, contact your HP representative.

```
W Object file is named $ vol. subvol.ZZBI nnnn
```

Binder assigned an arbitrary name after other file-naming attempts failed for the target file. nnnn is a numerical suffix that uniquely identifies the file. vol and subvol are the same as would have been used. This is an informative message only.

1

```
E ADD/REPLACE/DELETE may not be used on OWN (SAVE)
  blocks
```

You cannot specify an OWN (SAVE) block separately from its associated code block. Adding, replacing, or deleting the code block automatically adds, replaces, or deletes the own block.

2

```
E ADD/REPLACE/DELETE may not be used on PUCB blocks
```

These commands cannot refer to a COBOL85 PUCB by name. Although a COBOL85 PUCB is not classified as an OWN block, Binder treats it the same as an OWN block in ADD, REPLACE, and DELETE commands

3

```
E ALTER specification reused
```

You cannot specify an ALTER option more than once in a command line.

4

```
W Alter to CALLABLE and not PRIV means CALLABLE and
  PRIV: entry-point-name
```

Callable procedures must also be privileged code; Binder automatically sets PRIV ON if you specify CALLABLE ON. PRIV OFF is overridden in this case.

5

```
E Bad object file format version: file-name
```

The object file must be in the standard Binder format. If the code was produced by an earlier, incompatible version of the compiler, you must recompile. (An older version of Binder also displays this message if it attempts to process a new object file produced by a later version of Binder.)

6

```
E Block does not exist in file: block-name
```

No code or data block by the given name was found in the specified object file. You can check block names by using the LIST LOC command or by referring to a current compiler listing. You can ensure that Binder searches the correct file by giving the file name on the command or by using the FILE command.

7

```
E Part of data block being dumped is not initialized
```

The range of the data block specified in the DUMP or MODIFY command is not completely initialized. Binder compresses uninitialized data out of the target file. Therefore, it cannot be displayed.

8

```
E Block is not on the include list
```

The block specified in a MODIFY command is not in the include list (code or data) indicated by the command.

9

```
W Block length/address mode error on
  data-block-name

Additional text is one of the following:

  Blocks compared are from filename
  and filename
  Blocks compared are from filename
  and the compiler
```

The CHECK BLOCK parameter (SELECT command) is set ON, and common or global declarations are inconsistent in length or addressing mode. The data block reference is unresolved. You can correct the inconsistency or use SELECT CHECK BLOCK OFF to allow resolution. (The INFO UNRESOLVED * command displays unresolved code and data blocks.)

The additional text indicating which blocks are compared can have two forms. If you are using Binder interactively, the text indicates which object files Binder gets the data blocks from. If you are compiling a program, the text indicates which object file contains the data block that Binder compares with the compiler definition.

10

```
E Cannot add entry point to omit list if already on
  include list: entry-point-name
```

An entry point cannot be on both lists. If the include list is wrong, use the DELETE command. If the omit list is wrong, use the RESELECT OMIT command to clear the list.

11

```
E Can only ALTER MAIN attribute of C or TAL procedure
```

You cannot change the MAIN attribute after compilation for COBOL85, FORTRAN, and Pascal code blocks. (Verify that the ALTER LIKE command is used only if both code blocks have the same MAIN characteristic.)

12

```
E Cannot satisfy references since include lists are
  empty
```

You must add code and data block names and entry point names to the include lists before you enter a SATISFY or BUILD command. If you enter a BUILD command first, there are no names on the lists for Binder to include in the target file. Additionally, Binder clears all previous specifications set by commands such as SELECT, SET, and FILE. You must reenter those specifications.

13

```
W Cannot create file, using OBJECT instead:
  file-name ( error-num)
```

An attempt to use `file-name` as the name of the target file failed. Binder uses the name OBJECT instead. `error-num` is the file error received. Refer to the *Guardian Procedure Errors and Messages Manual* for information on `error-num`.

14

```
E\W Cannot create file: vol.subvol.OBJECT
    ( error-num)
```

The attempt to use OBJECT to name the target file failed. `error-num` is the file error received. Binder then attempts to create the target file with a name of the form ZZBI*nnnn*. If the attempt fails, the target file definition is lost and, in interactive mode, Binder prompts for input. Refer to the *Guardian Procedure Errors and Messages Manual* for information on `error-num`.

15

```
E CHANGE file cannot be open (use CLEAR command or close the file)
```

Because the CHANGE command writes to the object file, the object file must not be open when this command is executed. Binder might be using the file, in which case the CLEAR command closes it.

17

```
E Code space overflow in PROC: block-name
```

The program exceeded the maximum amount of code space available. Use the COMPACT directive to instruct Binder to compact the object file by rearranging code blocks. Alternatively, you can use the SET USERLIBRARY ON command to change the number of code segments allowed from 16 to 32.

18

```
W Control data space overflow
```

This error occurs on COBOL85 input files.

The largest possible configuration exceeds available memory. Every file opened in a COBOL85 program has its buffers allocated dynamically in the upper 32K of memory. Because Binder cannot tell which files will be open concurrently, it assumes the worst case. The receive-control table and reply-message table are also allocated dynamically in the upper 32K of memory. The length of the latter is governed by the table length and sync depth specified in your Receive-Control paragraph. Reduce the number of concurrently open files, or reduce sync depth or table length, or both.

19

```
E Data block cannot be allocated: block-name
  Data block trying to fit space-location
```

No room is available for the data block in the correct part of memory for the block type. Reorganize the data space to make room for the specified block. space-location is one of the following:

FIXED POSITION	BELOW 32K LAST	ANYWHERE
BELOW 256	ABOVE 32K LAST	EXTENDED ADDRESS
BELOW 32K	BELOW 32K PENULT	
ABOVE 32K	BELOW 64	

20

```
E Data reference failed due to relocation:
  ( code-block-name) + ( offset)
  REF TO ( data-block-name) + ( offset)
```

This error occurs on TAL input files.

A BLOCK global variable can be moved past TAL limit. code-block-name contains the data reference. Rearrange global variables or use the INHIBITXX directive. Refer to the *TAL Reference Manual* for information on the INHIBITXX directive.

21

```
E Effective input record is too long
```

The total command line cannot exceed 528 characters, excluding continuation characters. Respecify as more than one command.

22

```
W Code block already on include list:
  entry-point-name
```

Your ADD command named a code block that is already on the include list. An entry point name cannot be on the include list more than once. If two object files contain the same code block name, Binder uses the first occurrence encountered. This may or may not be in the first file on the search list. For the rules governing the resolution of unresolved references, see Unresolved Reference Lists on page 5-6.

24

```
W Entry point already on include list:
  entry-point-name
```

The entry point was already inserted in the include list. If the second occurrence is the correct one, use the REPLACE command to replace the first occurrence. If you are replacing an SQL code block, see Binding SQL Program Files on page 2-14.

25

```
W Entry point cannot be resolved due to conflict
  with another procedure: entry-point-name
```

The entry point has also been used as a secondary entry point name in another code block that is already on the include list. Use the INFO INCLUDE * command to display all instances of the multiply defined code block.

26

```
W File already on search list file-name
```

The file name was previously used in a SELECT SEARCH command. The file name is ignored.

27

```
E File code not 100: file-name
```

The indicated file is not an object file.

28

```
E Identifier too long
```

An identifier cannot exceed 31 characters.

29

```
E Illegal ALTER PROC name: proc-name
```

The entry point name given as an ALTER LIKE entry point is not on the include list.

30

```
E Illegal block range member: name OR *
```

If you specified the * option, the include list is empty; otherwise, the range specifies a block name that is not on the include list. Use the INFO INCLUDE * command to find out which name is not on the list.

31

```
E Illegal DUMP offset(s)
```

The offset and length specified in the DUMP command exceeds the block length. Reenter the command with the * option or with the correct length and offset.

32

```
E Illegal DUMP specification
```

You cannot use ICODE dump specification for a data block.

33

```
E Illegal log file - ignored
```

You used the LOG file name for one of: IN file, OUT file, or OBEY file.

34

```
E Illegal MOVE PROC name: code-block-name
```

A MOVE command cannot refer to a block that is not on the include code block list nor to a block that is inside a range of blocks to be moved.

35

```
E Illegal OBEY file - ignored
```

You used the OBEY file name as one of: IN file, OUT file, or LOG file.

36

```
E Illegal offset for MODIFY/VERIFY command
```

You specified an offset for the MODIFY or VERIFY command that extends beyond the end of the block.

37

```
E Illegal OUT file - ignored
```

You used the OUT file name as one of: IN file, OBEY file, or LOG file name.

38

```
E Illegal SELECT REFER pair - ignored
```

Names of a refer pair cannot be the same: a new name cannot have been used as the old name of another refer pair; likewise, the old name cannot be the new name of another refer pair.

39

```
E Illegal SET value - ignored
```

The maximum values for SET parameters are:

PEP	512
DATA	64 pages (65536 words)
EXTENDSTACK	same as DATA
STACK	same as DATA
RESERVE BELOW64	124 bytes
RESERVE PRIMARY	508 bytes
RESERVE SECONDARY	65,024 bytes
RESERVE EXTENDED	1,073,741,824 bytes

40

```
E Integer conversion error
```

Either the number supplied is too large or an invalid digit was received.

41

```
E Invalid file name file-name
```

The file name does not conform to operating system requirements.

For a description of the file naming conventions, see Section 7, Guardian File Names and TACL Commands.

42

```
E Invalid subvolume name
```

The volume or subvolume name is too long or contains an invalid character.

For a description of file naming conventions, see Section 7, Guardian File Names and TACL Commands.

43

```
E Invalid syntax
```

The sequence of input characters does not result in a valid Binder command. A circumflex (^) indicates the detected error.

44

```
W Invalid system name
```

The system name is too long or contains an invalid character, or the system does not exist. For a description of the file naming conventions, see Section 7, Guardian File Names and TACL Commands.

45

```
W MAIN entry point found in a search file: proc-name
```

In attempting to include a procedure from a search file (because it has been directly or indirectly referenced by the program being compiled), BINSERV has discovered that the procedure has the MAIN attribute. You can remove the MAIN attribute with the ALTER command.

46

```
W MODIFY overrides reference to another block in:
  block-name
```

The MODIFY command resulted in a change to a CALL or to a reference to global data. This message is issued both at the time of the MODIFY and when the block is added to an include list in a later Binder session.

48

```
E More than 512 entry points in the XEP table,
  nnn: in space % space-number
```

There are more than 512 entry points in the external entry point table for the indicated code space. Some of these unresolved references must be satisfied before the object file can be built.

49

```
E Multiple MAIN entry points are not allowed:
  entry-point-name
```

Only one entry point can have the MAIN attribute.

50

```
E No current file for ADD/REPLACE/LIST/DUMP
```

You must establish a current file before the named commands can be executed. Either reenter the command with the FROM *file-name* parameter or use the FILE command. FILE establishes the default file for subsequent commands (or until another FILE is entered).

51

```
E No help available for name
```

You have specified a command parameter in a form that is not recognizable by the HELP command. Use HELP *command-name* to verify the parameter forms. HELP with no parameters lists all the Binder commands.

52

```
W No parameter information provided for entry-name
```

This error occurs on COBOL85 input files.

Binder encountered a CANCEL for the entry point name, but no CALL statement referred to the entry point. Therefore, parameters could not be checked.

53

```
E No space left for stack after data block
  allocation
```

Word 32767 is used in the word-addressable data space. Binder cannot allocate the stack. Reorganize the data space.

54

```
E Not enough space for XEP table; n words
  required, m words available
```

The external entry point table cannot be placed in the code space.

55

```
E OBEY nesting exceeds maximum
```

Three levels of nesting is the maximum allowed for OBEY files.

56

```
E No BINDER region in object file: file-name
```

This error can occur for two reasons:

1. The file had Binder tables but has been stripped by a Binder STRIP command. Binder cannot manipulate the file in any way; recompile.
2. The file was compiled by a compiler version that does not incorporate Binder. Recompile using the correct version. Check with your system support personnel.

57

```
E Incompatible version of INSPECT information for
  entry-point-name
```

Symbol table incompatible with the current version of Inspect. Recompile the program containing *entry-point-name*.

58

```
F Paging file error file-err-msg ( nnn) or
  ALLOCATESEGMENT err-code nnn, volume: $ vol-name
```

Refer to the *Guardian Procedure Errors and Messages Manual* for a description of the indicated error type.

59

```
W Parameter count mismatch on entry-point-name
```

SELECT CHECK PARAMETER is STRICT (default), and the consistency check failed. The number of parameters in the call differ from the number required by the named entry point. Binder also issues this message if calls to the same entry point from separately compiled code blocks specify a different number of parameters.

60

```
W Parameter mode mismatch on entry-point-name
  parameter n
```

SELECT CHECK PARAMETER is STRICT (default), and the consistency check failed. The mode of the indicated parameter differs from that required by the named entry point. Parameter mode is by value, by reference, or extended reference. Binder also issues this message if calls to the same entry point from separately compiled code blocks specify different parameter modes.

61

```
W Parameter type mismatch on entry-point-name
  parameter n
```

SELECT CHECK PARAMETER is STRICT (default), and the consistency check failed. The type of the indicated parameter differs from that required by the named entry point. Binder also issues this message if calls to the same entry point from separately compiled code blocks specify incompatible types.

Binder goes on to say whether the mismatch was between two external declarations of a procedure or between an external declaration and the actual procedure declaration. Binder detects when at least one parameter is of type INT, INT(32), or STRING and informs you.

If only one parameter is INT, INT(32), or STRING, Binder designates the parameter that is not one of those types as "other." If neither parameter is INT, INT(32), or STRING, Binder outputs more detailed information on the mismatch, including one or more of the following: parameter storage type, parameter length in bits, number of units, and number of digits to the right of the decimal point.

62

```
E Primary global area overflow
```

This error can occur with FORTRAN or TAL input files.

For FORTRAN, this error means that too many variables in FORTRAN COMMON are referenced. Recompile using the ?EXTENDCOMMON compiler directive.

For TAL, this error means that too many global variables are defined. Modify your global data space.

63

```
W Procedure references absolute global data which
  may be relocated: code-block-name
```

This error occurs on TAL input files.

The RELOCATE directive was used and relocation might have occurred. The #GLOBAL block is not at offset zero. The compiler and command-driven Binder issue warnings.

64

```
E Range members in wrong order: name to
  name
```

If the ADD or REPLACE command has an incorrect range given, respecify block names in ascending order by location in the object file. Use LIST LOC command or a current map listing to verify the order of blocks in the input file.

If the command refers to a range in an include list, respecify the command with names in ascending order as on the list. You can use the INFO INCLUDE command.

65

```
W Read-only data block moved to above 32K to avoid
  straddling 32K: data-block-name: in space
  % space-number
```

This error occurs on TAL input files. A string P-relative array was moved.

66

```
W Reference to string P-relative in wrong half of
  code space may fail: proc-name +
  offset REF TO block-name + offset
```

This error occurs on TAL input files.

Binder cannot verify that references to the array are valid. Note that offset is the offset of a fix-up word.

67

```
E RENAME code block is not on the include list
```

The code block specified in a RENAME command must be on the include list.

68

```
E RENAME data block is not on the include list
```

The data block specified in a RENAME command must be on the include list.

69

```
W Return type mismatch on proc-name
```

SELECT CHECK PARAMETER is STRICT (default), and the consistency check failed. The return type required by the named procedure is inconsistent with the call. This message is also issued if calls to the same entry point from separately compiled code blocks specify incompatible return types. You can disable parameter checking by setting CHECK PARAMETER OFF.

70

```
F Storage file error file-sys-err ( nnn)
```

An error occurred in the temporary work file that Binder uses to hold code and data until the object file is built. Refer to the *Guardian Procedure Errors and Messages Manual* for a description of the error.

71

```
E STRIP command cannot be used on this file
```

The STRIP command cannot be used on a valid old-format object file.

72

```
E STRIP file cannot be open (use CLEAR command)
```

Binder is currently using the file for prior commands (for example, FILE). CLEAR resets Binder to the initial state. If no other opens are current for the file, it can then be stripped.

73

```
F Symbol file error: file-sys-err ( nnn)
```

An error occurred in the temporary work file that Binder uses to hold symbol information until the object file is built. Refer to the *Guardian Procedure Errors and Messages Manual* for a description of the error.

74

```
E The MAIN must be in COBOL since COBOL procedures
  are present
```

Mixed-language binding of C-series programs with COBOL85 code requires that the MAIN program be COBOL85 to allow correct initialization for the COBOL85 run-time

environment. Consider adding a skeleton program that contains a call to the major entry point. (Only one block with the MAIN attribute is allowed.)

75

```
W TNS/II user library violation: MAIN procedure
  code-block-name
```

SELECT CHECK LIBRARY ON is in effect and a procedure with the MAIN attribute was encountered.

76

```
W TNS/II user library violation: referencing data
   block block-name
```

A data block with read-write access was added to the include list and CHECK LIBRARY ON is in effect. The addition of this block can affect data memory organization.

77

```
E Unsatisfied reference to a data block
```

Before you enter a BUILD command, the unresolved data reference list must be empty. There are still unresolved data block references. Use INFO UNRESOLVED DATA command to check the list.

78

```
E Unterminated continuation line
```

End of file was encountered while scanning for the remainder of a continuation line.

79

```
E Unterminated string
```

The ASCII input to a MODIFY command is missing the closing quotation mark.

80

```
F Value specified in VERIFY command not equal to
  current value: should be % value, is % value
```

You can use the VERIFY command during a noninteractive BIND session to stop the session if a discrepancy exists between an expected code or data value and the actual value. This message indicates which value did not match the expected value.

81

```
W Extensible or variable attribute mismatch on
  proc-name
```

This error occurs on TAL input files.

Different callers disagreed about the VARIABLE or EXTENSIBLE attribute of an entry point.

82

```
F Work file error: file-sys-err ( nnn)
```

The work file is the object file being constructed. Refer to the *Guardian Procedure Errors and Messages Manual* for a description of the error type.

83

```
W Old object file has been renamed to file-name
```

This warning indicates the new name of a file that was already in use when you tried to perform a Binder operation on it. The renaming permits the user to name the new object file. If the old file is not in use, it is purged. If the old file is in use, it is renamed.

84

```
F The compiler has used the same unique ID for two
  different names
```

This message indicates a Binder internal error. Contact your HP representative.

85

```
F An OWN block or COBOL PUCB has been included in
  the object file without the associated code for
  the data block: block-name
```

When you include an OWN block or COBOL85 PUCB in an object file, you must also include the associated code for the data block.

86

```
W Reference to a block expected, but not in,
  G-relative address area block-name +
  offset REF TO block-name + offset
```

Binder was expecting a G-relative address reference for a block but received an address outside of G-relative space.

87

```
F The attributes of two FCBs for the same data block
  do not match block-name
```

The attributes of FCBs for the same data block must match. This could be a user error but is more likely a compiler error. Contact your HP representative.

88

```
E There is too much data for the largest extended
  data segment allowed
```

You have added more extended data in extended data blocks than is allowed in the largest extended data segment. Use less extended data.

90

```
E Block being renamed or block renaming to is a
  special data block
```

You cannot use a RENAME command to change the name of a special data block or to change a block name to that of a special data block name.

91

```
W OWN data blocks contained in the range of blocks
  were not added
```

OWN data blocks cannot be contained in a range of data blocks being added to the object file. You can only add OWN data blocks by adding the code block to which they belong. All data blocks other than the OWN block were added.

92

```
E Possible OWN block name conflict with a COMMON
  block: block-name
```

You attempted to bind two data blocks that are of different types but that have the same name. Resolve the name conflict and recompile.

93

```
E Insufficient stack space for program, unable to
  create object file
```

The data below 32K words in the data space plus the amount of stack space needed for the program to execute add up to more than 32K words. You must either reduce the data blocks below 32K words or reduce the amount of stack space used.

94

```
E Block being renamed or block renaming to is an
  indirect data block
```

You cannot use a RENAME command to change the name of an indirect data block or to change a block name to that of an existing indirect data block.

95

```
E Block being renamed to is already on include list
```

The data block name on the right side of the RENAME command already exists on the include data list.

97

```
E A space number less than 0 or greater than %37
  has been used
```

Code segment numbers outside the range of 0 through %37 are not valid segment numbers.

98

```
E A space number has been given with a data address
```

Data blocks do not have a code segment number associated with them. Remove the code segment from the command, and try again.

99

```
E Illegal SUBTYPE value -- ignored
```

The SUBTYPE value in the SET command must be in the range 0–63.

100

```
E More than 8160 entry points on include list:
```

The procedure entry point table can have a maximum of 8160 entries, one for each entry point. You must restructure the program.

101

```
E Parameter addressing incompatibility
```

You are attempting to bind a large model program with a small model program. This is not allowed. All code and data blocks included at the same time must be from the same model.

102

```
F Fatal trap
```

The Binder has failed with a trap; notify your HP representative.

104

```
W Illegal volume name passed in param swapvol
  message -- ignored
```

The volume name used in PARAM swapvol is not a valid volume name on your system.

106

```
W The Binder session contained errors or warnings (the error/warning
  counts exclude those found in the Build phase of the Bind session).
```

A noninteractive BIND session that has errors or warnings outside of the BUILD command has statistics printed at the end of the BIND session.

107

```
E ADD/ALTER/DELETE/MOVE/NAME/REPLACE not allowed on
  contained procedures
```

You cannot use these commands on procedures with a lexical level greater than one (nested code blocks).

108

```
E OMIT/REFER not allowed on contained procedures
```

You cannot use these commands with nested code blocks (procedures with a lexical level greater than one).

109

```
E MOVE not allowed between contained procedures
```

You cannot move a set of procedures between two procedures with a lexical level greater than one (nested code blocks).

110

```
E Part of extended data block being dumped is not
initialized
```

The range of the extended data block being dumped is not completely initialized.

111

```
E The data block EXTENDED#STACK#POINTERS is required
  but not present
```

Binder has seen the special data block $EXTENDED#STACK but not the special data block EXTENDED#STACK#POINTERS. EXTENDED#STACK#POINTERS is required with $EXTENDED#STACK, and it must be added to the include list at the same time.

112

```
E A required data block is not present:
  block-name
```

The special data block *block-name* is required on the include list for Binder to be able to build on the object file.

113

```
W The object file is not runnable for the following
  reason(s): reasons
```

reasons is one or more of the following:

- No MAIN code block
- Fixups are not applied
- Undefined data blocks are referenced
- References from data to code are unresolved

Correct the problem indicated. SELECT FIXUP ON applies fixups to the target file; INFO UNDEFINED * displays the undefined data blocks; and INFO UNRESOLVED * displays the names of unresolved entry points and data blocks.

114

```
W The data pages needed is greater than the data
  pages requested
```

This message is displayed when the data pages requested for the object file is less than the data pages needed for the object file.

118

```
E System is unavailable
```

The specified system name exists, but this system is unavailable.

121

```
W SET CODE SEGMENT entered with more than one code segment
  present
```

On TNS systems, SYSGEN checks the size of files sent to Binder to ensure that Binder builds only one code segment at a time. On TNS/R systems, SYSGEN sends to Binder a large group of files to build a multi-segment system code and system library.

You can safely ignore this message during the SYSGEN phase of INSTALL. If you receive this message at any other time, contact your HP analyst.

143

```
W Variable attribute mismatch on C function name
```

The attributes of two C functions do not match. One is a variable and the other is not. Correct your code and recompile before attempting to bind the files again.

144

```
E Block is incompatible with run-time environment: block-name
```

Binder issues this error if you try to bind an file compiled for the OLD Binder group and a file compiled for the COMMON Binder group into one object file.

BINSERV also issues this error if you try to bind an OLD Binder group file or a COMMON Binder group file with an object file with a ENV NEUTRAL directive in its source text.

146

```
E Illegal PFS Size. Legal range is 64 pages to 512 pages
```

Specify a process file segment (PFS) size within the range of 64 to 512 pages.

148

```
W Referencing procedures claim that procedure procedure-name
  is written in a different language
```

Procedures that call *procedure-name* incorrectly specify the language of *procedure-name*.

149

```
W Referencing procedures do not agree on the language of
  procedure procedure-name
```

Procedures that call procedure-name do not agree on the language of *procedure-name*. When Binder determines the language of *procedure-name*, it issues warning 148 for referencing procedures that specify the incorrect language.

151

```
E The COMMON run-time data block #MCB has a new format, use a
  newer BIND
```

Your version of Binder does not recognize the format of data block #MCB used for the Common Run-Time Environment. Use a version of Binder that is compatible with the version of the compiler.

152

```
W Parameter return type mismatch on procedure-name
  parameter number
```

The return types of procedures passed to *procedure-name* do not match. *number* corresponds to the parameter order in the called procedure.

153

```
W Parameter mismatch using lenient parameter checking on
  procedure-name parameter number
```

Procedures that call *procedure-name* do not match the declared parameters of *procedure-name*.

154

```
E Unable to express default volume in network form
```

Binder does not recognize eight-character volume names on non-local nodes.

165

```
E TARGET types conflict: entry point entry-name from file is
  type1, the current setting is type2
```

Change the TARGET attribute on one of the files and try the operation again.

166

```
W SET attribute has been changed: from TARGET ANY to
  TARGET type
```

If you enter the command SET TARGET ANY, and then later add in a TNS procedure or TNS/R procedure, Binder changes the target processor setting appropriately and issues this warning message. type can be TNS or TNS/R.

167

```
E Illegal mix of WIDE and NOWIDE memory attributes in file:
  file-name
```

You cannot bind C programs compiled under the wide-data model with C programs compiled under the large-memory model or the small-memory model. Refer to the *C Reference Manual* for details on C memory and data models.

209

```
E Invalid current working directory
```

You specified an invalid directory for the CD command. An existing OSS directory must be specified in the CD command. The previous directory setting remains in effect.

213

```
W The SET USERLIBRARY option is obsolete
```

You built an object file using the new features of the SRL Binder and the USERLIBRARY option is set to build with the implied user library. Binder does not use the implied user library when building the object file.

214

```
E More that 512 external entry points are referenced as
  initialization data
```

Binder encountered more than 512 different external entry points that must be referenced as initialization data. Binder does not build the target codefile.

215

```
W The block length or address mode does not match the
  requirements for the imported block: block-name
```

While building an SRL application, Binder encountered a data block that is also exported by the library. The user code and user library block definitions do not match. Binder builds the target codefile. The application is not runnable with the specified library. Supply a correct library before running the application.

216

```
W There are unresolved references to data block block-name
```

When building an SRL application, some of the external variable references were not resolved. Binder continues to build the object file and include the missing blocks on the imported block list. The application is not runnable with the specified library. A new library must be supplied to declare the missing blocks.

217

```
W Exported data block block-name was never found
```

When building an SRL user library, Binder found a data block that was named in an EXPORT command, but was not found in any of the modules that make up this library. Binder builds the target codefile, but the data block is removed from exported block list.

218

```
E The location constraint on data block data-block-name is
  illegal in a UL codefile
```

When building a user library, Binder encounters a data block with a location constraint is not supported for user library codefiles. For example, a data block that is declared at address 0 is illegal in a user library codefile. Binder does not build the target codefile.

220

```
W Reserved value was insufficient for data-type data space .
  x bytes were reserved . y bytes must be reserved for
  library data
```

The size of space you specified in the RESERVE command is not large enough for global data space requirements of the current library. Binder builds the target codefile and the size of the reserved area is increased to the size needed.

222

```
W The application is not runnable with the specified import
  library due to the mismatches reported above
```

While building an SRL application, Binder encountered one or more of the mismatches described in Warning 215. Binder builds the target application, but the application codefile is not runnable with the current instance of the library. The mismatches must be resolved by rebuilding the application under the application and/or the library codefiles.

223

```
E Data area is full. Cannot allocate data block block-name
```

You specified a data area that cannot be allocated because the data area is full. Binder does not build the target codefile.

224

```
W The reserve size of x exceeds the maximum for data area
  data-type
```

You specified a SELECT RESERVE command with a value that exceeds the maximum for the specified data area. The invalid setting is ignored and the previous setting is used.

226

```
W Duplicate definition of data block data-block-name
```

When adding data from a module, a data block was encountered that has already been declared. This warning only occurs if the CHECK DUPLICATE BLOCK option is turned on. Binder continues to process the ADD command and build the target object file. The first data block definition is added to the target file and the duplicate is ignored.

228

```
E Illegal combination of IMPORT and EXPORT options
```

The IMPORT and EXPORT commands are mutually exclusive. The contradictory command is ignored.

230

```
E File is not a TNS object file: filename
```

You specified a filename that is not a TNS object file. The command is ignored.

231

```
W There are data block references to unresolved code blocks
```

References from data blocks to code blocks are unresolved at bind time. Binder generates fixups for these references, so the references can be resolved to the user library at run time.

232

```
W Modify location is uninitialized,target object may not contain new data
```

When the build command is issued, Binder applies the modifications from the modify list. If the requested data block is not currently initialized, it is not modified.

233

```
F Error encountered during timestamp conversion.
```

Binder abends when the DST table is obsolete.

234

```
W Cannot SQL compile a stripped object file.
```

Binder cannot SQL-compile an SQL object file, if the object file is stripped of the Binder region.

770

```
E message table \.$SYSTEM.SYSTEM.PDTERROR
```

Either Binder cannot use the message table \.$SYSTEM.SYSTEM.PDTERROR or the $SYSTEM.SYSTEM.PDTERROR file does not contain error messages for Binder.

Completion Codes

This subsection lists the completion codes returned by Binder. These codes are displayed at the end of a Binder process and explain why a process terminated.

Table 8-1. Binder Completion Codes (page 1 of 2)

Code	Meaning
0	The process terminated abnormally. No WARNINGS or ERRORS were issued.
1	This code can have either of the following meanings: One or more WARNINGS were issued in a session. Object file is complete. One or more ERRORS were issued in a noninteractive session. Object file is complete.
2	A FATAL ERROR occurred. No object code was produced.

Table 8-1. Binder Completion Codes (page 2 of 2)

Code	Meaning
3	The process terminated prematurely because of errors. No object code was produced, and ERRORS were issued in a noninteractive session. For example, this completion code is returned if Binder is unable to open a file.
5	The process terminated abnormally. No object code was produced.
8	A WARNING was issued because Binder had to rename the object file.

9 Syntax Summary

This section summarizes Binder commands and their syntax.

Table 9-1. Binder Command Summary (page 1 of 2)

Command	Description
ADD	Inserts new names or replaces old names on the include lists; Binder deletes replaced names.
ALTER	Changes attributes of entry points.
BUILD	Creates the target file.
CD	Specifies the default current working directory.
CHANGE	Patches the attribute values in an already-created object file.
CLEAR	Returns Binder to its original state without creating the target file.
COMMENT	Enters comments to appear in the output listing.
DELETE	Removes names from include lists and removes associated changes from the modify list.
DUMP	Displays all or part of the contents of a code or data block.
ENV	Displays the current settings of the process environment controls.
EXIT	Stops the Binder process.
FC	Displays the previous command line, which you can then repeat or modify.
FILE	Sets the default object file for the ADD, DUMP, LIST, REPLACE, and SHOW commands.
HELP	Displays the Binder commands and syntax.
INFO	Displays information about code blocks, entry points, and data blocks on the include, unresolved reference, and undefined lists.
LIST	Specifies options for load maps and cross-reference listings.
LMAP	Displays an alphabetical load map for a specified file.
LOG	Starts or stops the recording of Binder input commands and output.
MODIFY	Changes the values of code or data block contents in the target file.
MOVE	Reorders code blocks on the include code block list.
OBEY	Directs Binder to read commands from the named disk file.
OUT	Names the file to receive output listings.
RENAME	Renames a code or data block.
REPLACE	Inserts replacements for code or data blocks on the include lists.
RESELECT	Resets one or more SELECT command parameters to the default value.
RESET	Restores one or more object file attributes to the default values.
SATISFY	Attempts immediate resolution of all external references.

Table 9-1. Binder Command Summary (page 2 of 2)

Command	Description
SELECT	Sets options for Binder operation control.
SET	Sets object file characteristics to use in building the target file.
SHOW	Displays collected information: current file, modify list, and controls from the SELECT and SET commands.
STRIP	Deletes Binder, Inspect, and Accelerator regions from the object file.
SYSTEM	Sets the default node name for expanding file names.
VERIFY	Verifies a code or data value in an object file.
VOLUME	Sets the default volume and subvolume for expanding file names.

```
ADD { CODE entry-list } [ FROM file-name ] [ , DELETE ]
    { DATA block-list }
    { *               }
or
ADD SPACE
```

```
ALTER entry-list , alter-spec [ , alter-spec ] ...
```

```
BUILD [ / OUT file-name / ] [ file-name ] [ ! ]
      [ , { set-param | select-param } ] ...
```

```
CD { directory }
```

```
CHANGE
{ AXCEL ENABLE { ON | OFF } } IN file-name
{ DATA value [ PAGES | WORDS | BYTES ] }
{ HIGHPIN { ON | OFF } }
{ HIGHREQUESTERS { ON | OFF } }
{ INSPECT { ON | OFF } }
{ LIBRARY file-name }
{ PFS value [ PAGES | WORDS | BYTES ] }
{ RUNNAMED { ON | OFF } }
{ SAVEABEND { ON | OFF } }
{ SYSTYPE { GUARDIAN | OSS } }
{ SUBTYPE number }
{ TARGET { TNS | TNS/R | ANY } }
```

```
CLEAR
```

```
COMMENT [ text ]
```

```
DELETE { CODE block-list }
       { DATA block-list }
       { *               }
```

```
DUMP [ / OUT file-name / ] { CODE code-block-name }
                           { DATA data-block-name }
{ offset [ , count ] } [ spec-list ] [ FROM file-name ]
{ offset [ , *     ] }
{ *                  }
```

```
ENV [ LOG       ]
    [ MODE      ]
    [ SYSTEM    ]
    [ VOLUME    ]
    [ DIRECTORY ]
```

```
EXIT
```

```
FC
```

```
FILE file-name
```

```
HELP [ / OUT file-name / ] [ topic [ subtopic [ subtopic ] ] ]
                           [ subtopic                        ]
                           [ < param-name >                  ]
```

```
INFO [ / OUT file-name / ]
{ INCLUDE { CODE block-list } [ , DETAIL ] }
{         { DATA block-list }              }
{         { ENTRY entry-list}              }
{         { * }                            }
{                                          }
{ UNRESOLVED { DATA }                      }
{            { ENTRY }                     }
{            { * }                         }
{                                          }
{ UNDEFINED *                              }
{                                          }
{ * [ , DETAIL ]                           }
```

```
LIST [ / OUT file-name / ]

{ { SOURCE                            } [ FROM file-name     ]      }
{ {                                   }                             }
{ { CODE name-list [ IN SPACE num ]   }                             }
{ {                                   }                             }
{ { CODE block-list                   }                             }
{ {                                   }                             }
{ { DATA name-list                    }                             }
{ {                                   }                             }
{ { DATA block-list                   }                             }
{ {                                   }                             }
{ { XREF [ XREF- options          ]   }                             }
{                                                                   }
{ ( list-option [ , list-option ] ... )                             }
{ [ FROM file-name ] [ , BRIEF ]                                    }
```

```
LMAP [ / OUT list-file / ] FROM file-name
```

```
LOG { TO file-name }
    { STOP         }
```

```
MODE { UPSHIFT | NOUPSHIFT }
```

```
MODIFY { CODE code-block-name }
       { DATA data-block-name }
[ modify-spec ] [ offset ] [ , value ]...
```

```
MOVE entry-list { AFTER entry-name  }
                { BEFORE entry-name }
                { IN NEW SPACE      }
     [ , entry-list { AFTER entry-name  } ]...
                    { BEFORE entry-name }
                    { IN NEW SPACE      }
```

```
OBEY [ / OUT file-name / ] file-name
```

```
{ OUT file-name                       }
{ command / OUT file-name / param-name }
```

```
RENAME { CODE entry-point-name } TO name
       { DATA data-block-name }
```

```
REPLACE { CODE entry-list } [ FROM file-name ]
        { DATA block-list }
        { *               }
```

```
RESELECT { select-param [ , select-param ] ... }
         { *                                   }
```

```
RESET { set-param [ , set-param ] ... }
      { *                             }
```

```
SATISFY { select-param                             }
        { ( select-param [ , select-param ] ... ) }
```

```
SELECT { select-param [ , select-param ]... }
select-param can be any of the following:

{ CHECK check-option }
{ CHECK ( check-option [ , check-option ]... ) }

COMPACT { ON | OFF }

COMPRESS DATA { ON | OFF }

FILESYS { OSS | GUARDIAN }

FIXUPS { ON | OFF }

{ LIST listing-option }
{ LIST ( listing-option [ , listing-option ]... ) }

{ OMIT entry-name }
{ OMIT ( entry-name [ , entry-name ]... ) }

{ REFER refer-pair }
{ REFER ( refer-pair [ , refer-pair ]... ) }

RUNNABLE OBJECT { ON | OFF}

SATISFY { ON | OFF }

{ SEARCH file-name }
{ SEARCH ( file-name [ , file-name ]... ) }

WARNINGS { ON | OFF }
```

```
SET { set-param [ , set-param ]... }
set-param can be any of the following:
{ DATA        } value [ PAGES | WORDS | BYTES ]
{ EXTENDSTACK }
{ STACK       }

HEAP value [ PAGES | WORDS | BYTES ]

HEAP STATISTICS { ON | OFF }

HIGHPIN { ON | OFF }

HIGHREQUESTERS { ON | OFF }

IMPORT DATA variable-name-list

INSPECT { ON | OFF }

LARGESTACK value [ PAGES | WORDS | BYTES ]

LIBRARY file-name

LIKE file-name

PEP value

PFS value [ PAGES | WORDS | BYTES ]

RESERVE { BELOW64|PRIMARY|SECONDARY|EXTENDED } [ +|- ] nb

RUNNAMED { ON | OFF }

SAVEABEND { ON | OFF }

SUBTYPE number

SYMBOLS { ON | OFF }

SYSTYPE { OSS | GUARDIAN }

TARGET [ TNS | TNS/R | ANY ]

USERLIBRARY { ON | OFF }
```

```
SHOW [ / OUT file-name / ]
{ AXCEL ENABLE [ FROM file-name ]  }
{ FILE                             }
{ IMPORT                           }
{ INFO [ FROM file-name ]          }
{ MODIFY                           }
{ RESERVE                          }
{ SELECT                           }
{ select-param                     }
{ SET attribute [ FROM file-name ] }
{ SET                              }
{ set-param                        }
```

```
STRIP file-name [ , SYMBOLS | , AXCEL ]
```

```
SYSTEM [ node ]
```

```
VERIFY { CODE block-name } [ verify-spec ] [ offset ] , value
       { DATA block-name }
```

```
VOLUME { $ volume             }
       { [ $ volume. ] subvol }
```

Glossary

absolute pathname. A pathname that begins with a slash (/) character and is resolved beginning with the root directory. Contrast with "relative pathname."

accelerate. To use the Accelerator program to generate an accelerated object file.

accelerated object code. The RISC instructions that result from processing a TNS object file with the Accelerator.

accelerated object file . The object file that results from processing a TNS object file with the Accelerator. An accelerated object file contains the original TNS object code, the accelerated object code and related address map tables, and any Binderÿ and symbol information from the original TNS object file.

Accelerator. A program that processes a TNS object file and produces an accelerated object file. Most TNS object code that has been accelerated runs faster on TNS/R processors than TNS object code that has not been accelerated. Accelerated object files run no faster on TNS/E machines than TNS programs that have not been accelerated, for TNS/E you must use OCA.

Accelerator region. The region of an object file that contains the RISC instructions generated by the Accelerator.

ASSIGN Command. A TACL command that lets you associate a logical file name with a physical file name. The physical file name is a fully qualified file ID. See also filename and file ID.

application program interface. The set of functions or procedures that permits user programs to communicate with the HP NonStop operating system.

API. See application program interface.

BIND. The stand-alone Binder you can use to bind separately compiled object files (or modules) into a new object file.

Binder region. The region of an object file that contains a header and the following Binder tables: procedure information table, entry point table, and data block information table.

binding. The operation of examining, collecting, linking, and modifying code and data blocks from one or more object files to produce a target object file.

BINSERV. The Binder that is integrated with the C, COBOL85, FORTRAN, and TAL compiler.

block. The smallest unit of code or data that can be relocated as a single entity.

code block. The smallest independently relocatable piece of a program. Code blocks contain executable machine instructions and possibly inline constant data. Compare with data block.

common data block. A data block with a scope defined as public to all modules.

Common Run-Time Environment (CRE). A set of services implemented by the CRE library that supports mixed-language programs. Contrast with language-specific run-time environment.

Common Run-Time Environment (CRE) library. A collection of routines that supports requests for services managed by the CRE, such as I/O and heap management, math and string functions, exception handling, and error reporting. CRE library routines can be called by C, COBOL85, FORTRAN, Pascal, and TAL user routines and run-time libraries.

compilation unit. A source file plus source code that is read in from other source files by SOURCE directives, which together compose a single input to the compiler.

compiler directive. A compiler option that lets you control compilation, compiler listings, and object code generation. For example, compiler directives let you compile parts of the source file conditionally or suppress parts of a compiler listing.

CRE. See Common Run-Time Environment (CRE)

Crossref. A stand-alone product that collects cross-reference information for your program.

data segment. A segment that contains information to be processed by the instructions in the related code segment. Applications can read and write to data segments. Data segments contain no executable instructions.

DEFINE command. A TACL command that lets you specify a named set of attributes and values to pass to a process.

data block. The smallest independently relocatable piece of a program. Data blocks contain statically allocated variables or constants. Compare with code block.

DLL. See Dynamic Linked Library.

Dynamic Linked Library. This is a library loadfile that has symbols that can be referenced by another loadfile to resolve symbolic references at link time or at runtime. This loadfile offers functions or data for use by other loadfiles. For TNS/E, DLLs replace SRLs commonly associated with the TNS/R architecture. The object file linker `eld` generates DLLs for TNS/E (as does `ld` for the TNS/R DLLs). In UNIX, this type of file is known as a shared object file or dynamic shared object (DSO).

entry point. A location where a code block can be accessed. See also primary entry point and secondary entry point.

extended data segment. A segment that provides up to 127.5 megabytes of indirect data storage. A process can have more than one extended data segment.

external entry point (XEP)table. The XEP table contains an entry for each unresolved external reference and is in the last page of each code segment.

file ID. The last of the four parts of a file name; the first three parts are node name (system name), volume name, and subvolume name.

filename. In the OSS environment, a component of a pathname containing any valid characters other than a slash (/) character or null. In the Guardian environment, the set of node name, volume name, subvolume name, and file identifier characters that uniquely identifies a file.

file system. In the OSS environment, a collection of files and file attributes. A file system provides the namespace for the file serial numbers that uniquely identify its files. See also ISO/IEC IS 9945-1:1990 (ANSI/IEEE Std. 1003.1-1990), Clause 2.2.2.38.

On an HP system, the Guardian file system for a node is a subset of OSS virtual file system and is therefore contained within a single fileset. Traditionally, the application program interface for file access in the Guardian environment is referred to as the Guardian file system.

Guardian. The original application program interface (API) to the HP NonStop Operating System.

Guardian services. An application program interface (API) to the HP NonStop Operating System and associated tools and utilities.

global data. The identifiers that are accessible to all compilation units in a binding session.

high PIN. A process identification number (PIN) that is greater than 255. Contrast with low PIN.

HP NonStop Operating System. The operating system for HP NonStop systems. The operating system does not include any application program interfaces.

input control lists. The lists Binder uses to determine which code blocks to include in the target file.

Inspect region. The region of an object file that contains symbol tables for all blocks compiled with the SYMBOLS directive or pragma. These tables are used by Inspect. Also called the symbol region.

Itanium instruction region. Register-oriented 64-bit machine instructions that are directly executed on TNS/E processors. IPF instructions execute on TNS/E systems, but not on TNS systems or on TNS/R systems. OCA-generated IPF instructions are produced by translating a TNS object file using OCA. Native-compiled IPF instructions are produced by compiling source code with a TNS/E native compiler.

local data. Data that you declare within a procedure; identifiers that are accessible only from within that procedure.

low PIN. A process identification number (PIN) in the range 0 through 254. Contrast with high PIN.

lower 32K-word area. The lower half of the user data segment. The global, local, and sublocal storage areas.

language-specific run-time environment. A set of services implemented by the run-time library of each language. Without the CRE, C, COBOL85, FORTRAN, Pascal or TAL programs each have their own language-specific run-time environments. These language-specific run-time environments are often incompatible with each other. Contrast with Common Run-Time Environment.

language-specific run-time library. A collection of routines outside the CRE that supports requests from a specific language for services such as I/O and heap management, math and string functions, exception handling, and error reporting.

large memory model. A program attribute that specifies that a program's heap is allocated in the extended memory segment.

main routine. The first routine to execute when a program is run. The main routine determines the run-time environment for a program. It is the routine declared with the MAIN or PROGRAM keyword.

mixed-language program. A program that contains source files written in different HP programming languages.

NonStop Operating System Open System Services (OSS). An application program interface (API) to the HP NonStop Operating System and associated tools and utilities.

object file. A file generated by a compiler or binder that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. The file may be a complete program that is ready for immediate execution, or it may be incomplete and require binding with other object files before execution.

Object Code Accelerator (OCA). A program that processes a TNS object file and produces an object file that contains both TNS instructions and Itanium instructions. Most TNS object files that have been processed by OCA run faster on TNS/E machines than do TNS object files that have not been processed by OCA.

own data block. A data block with a scope defined as private to one module or code block. Also referred to as an private data block.

PARAM command. A TACL command that lets you associate an ASCII value with a parameter name.

pathname. The string of characters that uniquely identifies a file within its file system. A pathname can be either relative or absolute. See also ISO/IEC IS 9945-1:1990 (ANSI/IEEE Std. 1003.1-1990 or POSIX.1), Clause 2.2.2.57.

PEP table. See procedure entry point (PEP) table.

PIN. See process identification number (PIN).

primary data space. The area of the user data segment that can store pointers and directly addressed variables.

primary entry point. The location where a code block can be accessed through a PCAL or XCAL instruction. The name of the primary entry point is the same as that of the code block.

private data area. The part of the data space that is reserved for the sole use of a procedure or subprocedure while it is executing.

private data block. A data block with a scope defined as private to one module or code block. Also referred to as an own data block.

procedure entry point (PEP) table. The PEP table contains the entry point addresses for each code block and is in the first page of each code segment.

process. A program that has been submitted to the operating system for execution. An instance of execution of a program.

process identification number (PIN). An unsigned integer that identifies a process in a processor module. Internally, a PIN is used as an index to the process control block (PCB) table.

program file. An executable object file. It must contain an entry point with the MAIN attribute.

public name. A specification within a TAL procedure declaration of a procedure name to use in Binder, not within the compiler. Only an EXTERNAL procedure declaration can include a public name. If you do not specify a public name, the procedure identifier becomes the public name.

relative pathname. A pathname that does not begin with a slash (/) character. A relative pathname is resolved beginning with the current working directory. Contrast with "absolute pathname."

RISC instructions. Register-oriented 32-bit machine instructions that are directly executed on TNS/R processors. RISC instructions execute only on TNS/R systems, not on TNS systems. Contrast with TNS instructions.

RTDU. See run-time data unit (RTDU).

run-time data unit (RTDU). Region of an object file used to store NonStop SQL source and object code.

run-time environment. The services provided by run-time library routines and data objects (data blocks and pointers) to a program at run-time.

run-time library. A collection of routines that supports requests for services such as I/O and heap management, math and string functions, exception handling, and error reporting.

secondary entry point. The location where a code block can be accessed through a PCAL instruction. Only FORTRAN and TAL routines allow secondary entry points. Secondary entry points have distinct names.

shared run-time libraries (SRL). A library that allows run-time linked libraries to contain global variables. Binder commands support shared global data between applications and library files. SRLs are replaced by DLLs in TNS/E.

single-language program. A program in which all routines are written in the same programming language.

small memory model. A program attribute that specifies that the program's heap is allocated in the user data segment.

source file. A file that contains source text such as data declarations, statements, compiler directives, and comments. The source file, together with any source code read in from other source files by SOURCE directives, compose a compilation unit that you can compile into an object file.

sublocal data. Data that you declare within a subprocedure; identifiers that are accessible only from within that subprocedure.

Symbol region. See Inspect region.

system. The processors, memory, controllers, peripheral devices, and related components that are directly connected together by buses and interfaces to form an entity that is operated as one computer.

TAL. See Transaction Application Language (TAL).

TNS/R. The HP computers that support the HP NonStop Operating System and that are based on reduced instruction set computing (RISC) technology. TNS/R processors implement the TNS/R instruction set and maintain application compatibility with TNS processors. The term TNS/R can refer to the instruction set, the architecture, or the processors.

TNS/R instruction. A 32-bit, register-oriented RISC machine instruction defined as part of the TNS/R environment. TNS/R instructions are implemented by the RISC chip of a TNS/R processor. These instructions execute on TNS/R systems but not on TNS systems.

TNS/E. 64-bit computers that support the HP NonStop operating system and that are based on INTEL's Itanium architecture. TNS/E machines are upwardly compatible with the TNS system-level architecture. Contrast with TNS and with TNS/R.

TNS/E instruction. See Itanium instruction.

TNS. The HP computers that support the Guardian operating system and that are based on the instruction set computing (CISC) technology. The term TNS can refer to the instruction set, the architecture, or the processors. Systems with these processor include the NonStop II, NonStop TXP, NonStop EXT, NonStop VLX, NonStop Cyclone, and NonStop CLX 600, CLX 700, and CLX 800 series.

TNS environment. The registers, instruction set, and processing logic that are defined by the TNS architecture.

TNS instruction. A 16-bit, stack-oriented machine instruction common to all TNS and TNS/R systems. On TNS systems, TNS instructions are implemented by microcode; on TNS/R systems, TNS instructions are implemented by millicode or by acceleration to RISC code.

TNS object code. The TNS instructions that result from processing source code with a TNS language compiler. TNS object code executes on both TNS and TNS/R systems.

TNS word. A 16-bit word. Named for the assumed operand size associated with most TNS instructions.

TNS object file. The object file created by a TNS compiler. The file contains TNS instructions and other information needed to construct the code spaces and the initial data for a TNS process.

Transaction Application Language (TAL). A systems programming language for NonStop systems.

upper 32K-word area. The upper half of the user data segment. You can use pointers to allocate this area for your data; however, if you use the CRE, the upper 32K-word area is not available for your data.

user data segment. An automatically allocated segment that provides modifiable, private storage for the variables of your process.

user library. A set of procedures that the operating system links to a program file at run time.

XEP table. See external entry point (XEP)table.

Index

Numbers

32K boundary 4-10

A

Accelerated mode
- executing in 1-7

Acceleration
- cross platform 1-8

Accelerator 1-8
- AXCEL ENABLE attribute 3-15
- introduced 1-5

Accelerator Region
- description 4-11
- stripping 3-75

ADD Command
- efficient usage 3-3
- examples 3-8
- syntax and description 3-6/3-8
- using search files 3-48
- with SQL subprograms 2-15

ADD SPACE Command 3-6
ADD, DELETE vs. REPLACE 2-15, 3-41
ALPHA option 3-30, 3-53
Alphabetic Load Maps 2-21, 2-24, 3-30, 3-32, 3-53
ALTER Command
- examples 3-10
- syntax and description 3-9/3-10

Amending attribute values 3-15
Arrays, global read-only 4-9
ASSIGN Command
- and TACL DEFINE name 7-9
- TACL Product 7-9

Attributes
- See also individual attributes
- amending or patching values 3-18
- amending values 3-15
- changing 3-9, 3-15/3-18
- clearing 3-18
- code block 4-1
- data block 4-4
- object file 3-11
- resetting 3-44
- setting 3-57/3-67
- showing 3-67/3-75
- target file 3-57/3-67, 5-8/5-12

AXCEL ENABLE attribute 3-15

B

BIND
- command file operation 2-2/2-3
- commands See Commands
- defined 1-2
- introduced 1-3
- manual operation 2-1

Binder
- BIND process 1-3
- BINSERV process 1-2
- commands See Commands
- cross-reference lists 2-25/2-26, 3-30, 3-54
- input object files 1-2
- languages used with 1-4
- listings 2-19/2-26
- load maps 2-21/2-25
- output object files 1-2
- prompt 2-1
- relation to other products 1-4
- starting
  - compiler-invoked 1-2
  - interactive 2-1
- stopping 3-23
- work files 2-17

Binder region

- description 4-12
- stripping 3-75

Binding 2-14, 3-50, 3-55, 5-11
- C 2-11
- changing the swap volume 2-17
- COBOL85 2-10
- compilation time 6-2
- C-Series object with D-Series object 2-6
- defined 1-1
- FORTRAN 2-10
- interactive 1-3, 2-2
- language checking 3-50, 3-55, 5-11
- logging a session 3-33
- mixed languages 2-13
- modules 2-5/2-17
- Pascal 2-12
- redirecting output 3-39
- resolving external references 2-16
- rules 2-6
- seperately compiled object files 2-5/2-16
- SQL subprograms 2-14/2-16
- target file specifications 2-4
- target file statistics 2-19

BINSERV 1-2

Blank common blocks 4-5

Blocks
- binding 2-5/2-17
- code and data 4-1/4-7
- defined 1-1
- deleting from include lists 3-19
- displaying content 3-20
- displaying information 3-25
- named 4-5
- read-only 4-1
- renaming 3-40
- replacing 3-41

Block-list 3-4

Block-name 3-4

Block-range 3-4

BRIEF option 3-31

BUILD Command
- examples 3-14
- SATISFY ON option 3-13
- setting parameters 3-49/3-57
- syntax and description 3-11/3-14

Building applications using SRLs 6-4

C

C
- binding 2-11
- blocks 4-5
- compressed data 3-52
- distinguishing character case 3-33
- memory models 2-12
- mixed language binding 2-13
- PARAM SWAPVOL command 2-18
- routine scope 4-3
- run-time libraries 2-11
- undefined list 2-4
- version used with Binder 1-4

CALLABLE attribute 3-10, 4-2

Calls from high-PIN requesters 3-16, 3-59

CD Command 3-14

CHANGE Command
- examples 3-18
- syntax and description 3-15/3-18

Changing working directory 3-14

Character case 3-33

CHECK option 5-11

CHECK PARAMETER option 3-55

CLEAR Command 3-18

CLIB 2-11

COBOL 1-8

COBOL85
- binding 2-10
- control blocks 2-10
- external records 4-5
- mixed language binding 2-13

procedure replacement 3-41
routine scope 4-3
user libraries 6-1
version used with Binder 1-4
Code area size 2-23
Code blocks
adding to include list 3-6
attributes 3-9, 4-2
cross-reference lists 3-30, 3-54
defined 2-21, 3-4
deleting 3-19
displaying content 3-20
displaying information 3-25
displaying multiply-defined 3-26
in multiple object files 2-6
load maps 2-21/2-24, 3-28
modifying word values 3-34
moving 3-36
names 4-1
name-lists 3-4
order 5-3
renaming 3-40
replacing 3-41
scope 4-3
See also Blocks
size 2-23
specifying target file order 2-4, 3-7
verifying word values 3-77
Code region 4-9
Code segment
boundary marker 3-6, 3-37
determining segments 4-9
determining size 3-63
load maps 2-23
Command files 2-2, 3-38
Commands
ADD 3-2, 3-6/3-9
ADD SPACE 3-6
ALTER 3-9
automatic file name expansion 3-3
BUILD 3-11/3-14
CD 3-14
CHANGE 3-15/3-18
CLEAR 3-18
COMMENT 3-19
correcting with FC command 3-23
DELETE 3-19
displaying parameters 3-24
DUMP 3-20
efficient usage 3-3
entering 2-1, 2-2
ENV 3-22
EXIT 3-23
FC 3-23
FILE 3-24
HELP 3-24
INFO 3-25/3-28
LIST 3-28/3-31
LMAP 3-32
LOG 3-32
MODE 3-33
MODIFY 3-34/3-36
MOVE 3-36
multiple-line 3-3
OBEY 3-38
order of 2-4
OUT 3-39
RENAME 3-40
REPLACE 3-41/3-43
RESELECT 3-43/3-44
RESET 3-44/3-46
SATISFY 3-47/3-49
SELECT 3-49/3-57
SET 3-57/3-67
SHOW 3-67/3-75
STRIP 3-75
summary 3-2
syntax for name lists 3-4/3-5

SYSTEM 3-77
VERIFY 3-77
VOLUME 3-78
Command-driven binding
command file 2-2
interactive 2-1
introduced 1-3
COMMENT Command 3-19
COMMON 2-7
Common data blocks 4-5
Common run-time environment 2-6, 2-13
COMPACT option 3-52, 5-11
Compilation time binding 1-2
Compiler directives 2-6
Completion codes 8-28
COMPRESS DATA option 3-52
Control blocks, in mixed binding 2-10
Control lists
commands to create 5-2
include code block 5-3
include data block 5-4
include entry point 5-4
include run-time data unit (RTDU) 5-5
modify 5-5
omit 5-5
refer 5-5
search 5-6
undefined 5-6
unresolved reference 5-6
CPU, specifying for compiler 7-7
CRE 2-6, 2-13
Cross platform acceleration 1-8
Crossref, relation to Binder 1-4
Cross-Platform Acceleration 1-8
Cross-reference lists
example 2-26
generating 2-25, 3-28, 3-54
CTRL/Y 2-2, 3-23
Current file
clearing 3-18
establishing 3-24
showing 3-67

D

DATA attribute 3-15, 3-58, 5-9
Data blocks
adding to include lists 3-6
attributes 4-4
common 4-5
cross-reference lists 3-30, 3-54
defined 3-6, 4-4
deleting 3-19
directly addressable 4-6
displaying 3-20, 3-25, 3-28
in multiple object files 2-5
indirectly accessed 4-6
load maps 2-24/2-25
modifying word values 3-34
name lists 3-4
name-lists 3-4
own 4-5/4-7
P-relative read-only data 4-6
renaming 3-40
replacing 3-41
See also Blocks
special 4-4
TAL example 4-6
verifying word values 3-77
Data region 4-11
Data space
allocating 3-15
specifying 3-58
Debugging
Crossref 1-5
Inspect 1-5, 3-16, 3-60
stripped files 3-32, 3-76
Default system, specifying 3-77
Default volume/subvolume, specifying 3-78
DEFAULTS DEFINEs 7-7

DEFINE command,TACL product 7-5
Defining target file 2-3
Definitions
    BIND 1-2
    binding 1-1
    BINSERV 1-2
    block 1-1
    code block 2-21
    data block 3-6
    entry point 2-21
    object file 1-1
    program 1-1
    target file 1-1
DELETE Command 3-19
Disk files
    names of 7-1
Disk space, managing 2-17
DUMP Command
    examples 3-22
    syntax and description 3-20/3-22

E

Efficient usage, BIND Commands 3-3
Entry points
    adding to include list 3-6
    changing attributes 3-9
    cross-reference lists 3-30, 3-54
    definition 2-21
    displaying 3-25, 3-28
    external 2-23, 4-10
    load maps 2-21/2-23
    name-lists 3-4
    primary and secondary 4-2
    scope 4-3
Entry-list 3-4
Entry-name 3-4
Entry-range 3-4
ENV Command 3-22
ENV directive parameters 2-6
Environment
    CRE 2-6
    language-specific 2-6
Environment parameters 3-22
Error checking 3-50
Error Messages 8-1
Execution modes
    accelerated 1-7
    TNS 1-7
Execution parameters
    clearing 3-18
    resetting 3-43
    showing 3-67
    specifying 3-49/3-57
    value during SATISFY 3-51
EXIT Command 3-23
EXTENDSTACK attribute 3-58, 5-9
EXTENSIBLE attribute 4-2
External call (XCAL) 4-1
External entry point (XEP)
    size 2-23
    table 4-10
External records 4-5
External references
    modifying 3-35
    resolving 2-16, 3-13, 3-47, 3-54
    to system procedures 2-17
    unsatisfied 2-16

F

FC Command 3-23
FCB (file control block) 2-10
FILE Command 3-24
File control block 2-10
FILE ID
    in file names 7-3
File names 7-1
    defaults in 7-3
    equating with logical file names 7-9
    internal 7-4

logical 7-4
parts of 7-2
replacing define names 7-5
specifying defaults 7-4
Files
Binder work files 2-17
changing attributes 3-15/3-18
establishing current 3-24
multiple-code segment 4-9
setting target file attributes 3-57/3-67
target attributes 5-9/5-11
FIXUPS option 3-4, 3-52
FLUT (FORTRAN logical unit table) 2-10
FORTRAN
blank common blocks 4-5
control blocks 2-10
logical unit table 2-10
mixed language binding 2-13
named blocks 4-5
procedure replacement 3-41
routine scope 4-3
version used with Binder 1-4
FORTRAN binding 2-10
FORTRAN, Pascal 1-8

G

Gap length 2-23
Global
names 4-3
P-relative length 2-23
read-only arrays in TAL 4-9
Global variables 6-4

H

HEAP attribute 3-59, 5-9
HEAP STATISTICS attribute 3-59
HELP Command 3-24
High PIN
determining 3-65
running 3-65
HIGHPIN attribute 3-15, 3-59, 5-9
rules for combining 3-41
HIGHREQUESTERS attribute 3-16, 3-59, 5-9
rules for combining 3-41

I

IMPORT DATA attribute 3-59
Include lists
clearing 3-18
code block 5-3
data block 5-4
deleting blocks 3-19
displaying contents 3-25
entry point 5-4
introduced 2-4
replacing blocks 3-41
run-time data unit (RTDU) 5-5
INFO Command
examples 3-27
syntax and description 3-25/3-27
Input control lists
clearing 3-18
include code block 5-3
include data block 5-4
include entry point 5-4
include run-time data unit (RTDU) 5-5
introduced 2-4
modify 5-5
omit 5-5
refer 5-5
search 5-6
undefined 5-6
unresolved reference 5-6
Input files
search efficiency 3-3
specifying 2-3
TACL command lines 2-2

Inspect
- attribute 3-16, 3-60, 5-9
- program 3-60
- region 4-11
- relation to Binder 1-5

Inspect region
- description 4-11
- stripping 3-75

Interative binding 1-3
Internal file names 7-4
INTERRUPT attribute 4-2

L

Language specific run-time environment 2-6
Languages used with Binder 1-4
LARGESTACK attribute 3-60, 5-9
LIBRARY attribute and option 3-16, 3-51, 3-60, 5-9, 5-12
Library calls 6-2
LIKE attribute 3-10, 3-60, 5-9
LIST Command
- examples 3-31
- syntax and description 3-28/3-31

LIST option 3-53
Listing options 3-53
Listings
- control 5-1/5-8
- cross-reference 2-25, 3-30, 3-54
- data block 2-24/2-25
- entry point maps 2-21/2-23
- generating 2-19, 3-28
- load maps 2-21/2-25
- target file statistics 2-19

LMAP Command 3-32
Load maps
- alphabetic 2-21, 2-24
- data blocks 2-24/2-25
- entry points 2-21/2-23
- for multiple code segments 2-23
- generating 3-53
- listing 3-28
- location 2-23, 2-25
- turning on/off 3-4, 3-53

LOC option 3-30, 3-53
Local storage, stack space 3-58
Location load maps 2-23, 2-25, 3-30, 3-53
LOG Command 3-32
Logical file names 7-4, 7-9
Low PIN
- determining 3-65
- running 3-65

M

MAIN attribute 3-9, 4-2
Maps
- code blocks 2-21/2-23
- data blocks 2-24/2-25
- entry point 2-21/2-23
- multiple code segments 2-23

Memory models, C 2-12
Messages 8-1
Millicode 1-6
MISALIGN 3-16, 3-60
Mixed language binding 2-13
- character case 3-33
- parameter checking 2-14

MODE Command 3-33
MODIFY Command
- examples 3-36
- showing modifications 3-67
- syntax and description 3-34/3-36

Modify list
- clearing 3-18
- described 5-5
- introduced 2-4

Modifying external references 3-35
Module declaration, in Pascal and C 4-3
MOVE Command
- examples 3-37

- syntax and description 3-36/3-37
Multiple code segments
- creating 3-8
- files 4-9
- load maps 2-23
Multiple-line commands 3-3

N

Named blocks 4-5
Names, private vs. public 4-3
Name-lists 3-4
Nested routines, referencing 4-3
NEUTRAL 2-7
Node name in file names 7-2
Nonexported, nonimported variables 4-6
Nonexternal items 4-6

O

OBEY 3-38
OBEY Command 3-38
Object Code Accelerator (OCA)
- accelerated mode 1-8
- explicit acceleration 1-8
- functional overview 1-6
- introduction 1-6
Object file
- accelerator region 4-11
- attributes
  - See Object file attributes
- binder region 4-12
- binding seperately compiled 2-6/2-10
- code region 4-9
- creating with multiple code segments 3-8
- cross-reference lists 2-25/2-27
- data region 4-11
- defined 1-1
- displaying content 3-20
- format 4-8/4-12
- inspect region 4-11
- load maps 2-21/2-25
- setting current file 3-24
- statistics 2-19
- structure 4-1/4-12
Object file attributes
- amending and patching 3-15/3-18
- AXCEL ENABLE 3-15
- clearing 3-18
- DATA 3-15, 3-58
- EXTENDSTACK 3-58
- HEAP 3-59
- HEAP STATISTICS 3-59
- HIGHPIN 3-15, 3-59
- HIGHREQUESTERS 3-16, 3-59
- IMPORT DATA 3-59
- INSPECT 3-16, 3-60
- LARGESTACK 3-60
- LIBRARY 3-16, 3-60
- LIKE 3-60
- PEP 3-60
- PFS 3-17, 3-61
- RESERVE 3-61
- RUNNAMED 3-17, 3-61
- SAVEABEND 3-17, 3-62
- setting 3-11, 3-57/3-67
- STACK 3-58
- SUBTYPE 3-17, 3-62
- SYMBOLS 3-62
- SYSTYPE 3-17
- TARGET 3-17, 3-62
- USERLIBRARY 3-63
OBJECT, as default file name 3-13
OCA ENABLE 3-16
OCA Translation Mode 1-8
OLD 2-7
Omit list
- clearing 3-18
- described 5-5

introduced 2-4
syntax 3-54
Online help 3-24
Optimization 1-6
OUT Command 3-39
Output listings
displaying 3-29
entry point maps 2-21/2-24
generating 2-19
load maps 2-21/2-25
specifying a file 2-2
statistics 2-19
Own data blocks 4-5/4-7

P

Page faults, reducing 3-37
PARAM BINSERV command 7-7
PARAM command, TACL product 7-7
PARAM SAMECPU command 7-7
PARAM SWAPVOL command 2-18, 7-8
PARAM SYMSERV command 7-8
Parameter checking 3-51, 3-55
mixed language bind 2-14
parameter checking 2-14, 3-50, 3-55, 5-11
Parameter checking, mixed language bind 2-14
PARAMETER option 3-51, 3-55, 5-12
Pascal
binding 2-12
blocks 4-5
compressed data 3-52
HEAP STATISTICS attribute 3-59
mixed language binding 2-13
PARAM SWAPVOL command 2-18
procedure replacement 3-7, 3-42
routine scope 4-3
run-time libraries 2-13
run-time library 2-13
undefined list 2-4
version used with Binder 1-4
Pascal HEAPUSED routine 3-59
PCAL (procedure call) 4-1
PDTERROR file 8-1
PDTHELP file 3-25
PEP
attribute 3-60, 5-9
length 2-23
table 4-10
PFS attribute 3-17, 3-61, 5-9
PIN
determining 3-65
setting 3-65
Primary entry point 4-2
PRIV attribute 3-10
Private
blocks 4-5
Private names 4-3
PRIVILEGED attribute 4-2
Procedure call (PCAL) 4-1
Procedure entry point
See PEP
Procedure length 2-23
Procedure replacement
ADD, DELETE command 3-6, 3-7
ADD, DELETE vs. REPLACE 2-14, 3-41
Pascal 3-7
REPLACE command 3-41
SQL subprograms 2-14/2-16
Process file segment
See PFS attribute
Process subtype, setting 3-62
Processes
named 3-17, 3-61
running at a high-PIN 3-18, 3-59
Processor, specifying 3-63
Program defined 1-1
Program environment parameters
displaying 3-22
ENV 3-22

LOG 3-32
MODE 3-33
SYSTEM 3-77
VOLUME 3-78
Program unit control block (PUCB) 2-10
Public names 4-3
PUCB(program unit control block) 2-10

R

Read-only arrays,TAL global 4-10
Read-only code segments 4-1
Read-only data blocks 2-24, 4-4
Redirecting output 3-39
Refer list
clearing 3-18
described 5-5
introduced 2-4
syntax 3-54
RENAME Command 3-40
REPLACE Command
efficient usage 3-3
examples 3-42
syntax and description 3-41/3-42
REPLACE command
with SQL subprograms 2-15
RESELECT Command 3-43
RESERVE attribute 3-61
Reserving space in applications 3-61, 6-5
RESET Command
examples 3-46
syntax and description 3-44, 3-45
RESIDENT attribute 3-10, 4-2, 5-11
Routines
nested 4-3
referencing 4-3
scope 4-3
RUCB (run-unit control block) 2-10
RUNNABLE OBJECT option 3-54
RUNNAMED attribute 3-17, 3-61, 5-9
rules for combining 3-41
Running processes at a high PIN 3-65
Run-time environment
definition 2-6
Run-time libraries
C 2-11
Pascal 2-13
Run-unit control block (RUCB) 2-10, 4-4

S

SATISFY Command
examples 3-48
interactive mode 3-48
setting parameters 3-49
syntax and description 3-47
SATISFY option 3-54
Save file, specifying 3-62
SAVE variables 4-6
SAVEABEND attribute 3-17, 3-62, 5-9
Scope
data blocks 4-4
routines 4-3
Search list
clearing 3-18
described 5-6
introduced 2-4
SEARCH parameters with SATISFY command 3-48
Searching for files, efficiency 3-3
Secondary entry point 4-2
SELECT Command
examples 3-57
syntax and description 3-49/3-57
SELECT parameters
defaults 3-43
resetting 3-43
setting 3-49/3-55
showing 3-67/3-70
SET Command
examples 3-65
syntax and description 3-57/3-65

SET parameters
- defaults 3-46
- resetting 3-44
- setting 3-57/3-63
- showing 3-67/3-70

Shared run-time libraries 6-4
SHOW Command
- examples 3-70
- syntax and description 3-67/3-70

Single-code segment files 4-9
Space
- reserving in applications 6-5

Special data block 4-4
SPOOL DEFINEs 7-6
SQL, binding subprograms 2-14/2-16
SRLs 6-4
STACK attribute 3-58, 5-9
Stack space 3-58
Static variables 4-6
Statistics of target file 2-19
Straddling 32K boundary 4-10
String 3-4
STRIP Command 3-75
Stripped files
- command 3-75
- debugging 3-76
- displaying load maps 3-32

SUBTYPE attribute 3-17, 3-62, 5-9
Subvolume name
- in file names 7-3
- specifying defaults 7-4

Swap volume, changing 2-18
Symbol tables 4-11
- retaining 3-62
- stripping 3-75

SYMBOLS attribute 3-62, 5-9
SYMSERV
- specifying which one 7-8

SYSTEM Command 3-77
System name
- in file names 7-2

System procedures, referencing 2-17
SYSTYPE attribute 3-17

T

TACL
- setting a PIN 3-65

TACL ASSIGN messages 2-2
TACL commands
- DEFINE 7-5
- PARAM 7-7

TACL DEFINE names, rules for 7-5
TAL 1-8
- data block example 4-6
- ENV NEUTRAL directive 2-6
- global read-only arrays 4-10
- routine scope 4-3
- user libraries 6-1
- version used with Binder 1-4
- $EXTENDED#STACK 3-60

TAL mixed language binding 2-13
TAPE DEFINEs 7-6
TARGET attribute 3-17, 3-62, 5-9
Target file
- building 3-11/3-14, 5-11
- code block order 2-4, 3-7
- cross-reference lists 2-25/2-27
- defined 1-1
- input control lists 2-4
- load maps 2-21/2-25
- modifying values of words 3-34
- moving blocks 3-36
- naming convention 3-13
- specifications 2-4
- specifying content 2-3
- specifying processor 3-63
- statistics 2-19
- structure 4-1/4-12

Target file attributes

changing 3-15/3-18
clearing 3-18
in built file 5-9/5-11
list of
See Object file attributes
resetting 3-44
setting 3-57/3-67
showing 3-67/3-75
Temporary files
specifying volume for 7-8
The Object Code Accelerator (OCA) 1-6
TNS mode, executing in 1-7
TNS processor 1-5, 3-63
TNS/R processor 1-5, 3-63

U

Undefined list
clearing 3-18
described 5-6
displaying contents 3-27
introduced 2-4
Unresolved external references, SATISFY Command 3-48
Unresolved reference list
clearing 3-24
described 5-6
displaying contents 3-26
introduced 2-4
User libraries 6-1
allocating space 3-63
binding procedures 6-1
command driven binding 6-2
compilation time binding 6-2
introduced 6-1
purpose 6-1
restrictions 6-3
specifying 6-3
USERLIBRARY attribute 3-63, 5-9

V

VARIABLE attribute 4-2
Variables 4-4
VERIFY Command
examples 3-78
syntax and description 3-77
VOLUME Command 3-78
Volume names
in file names 7-2
specifying defaults 7-4

W

Warnings 8-1
WARNINGS option 3-55
Words, modifying the values of 3-34
Work files, managing 2-17

X

XCAL (external call) 4-1
XEP
table 4-10
XEP (external entry point)
size 2-23
XREF option 3-30, 3-54

Z

ZZBInnnn 3-13

Special Characters

$EXTENDED#STACK 3-60
.(period)
in file names 7-2