DOMAIN Binder and Librarian Reference

Order No. 004977
Revision 02

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: August, 1984
First Revision: July, 1985
Second Revision: January, 1987

This document was produced using the Interleaf Workstation Publishing Software (WPS). Interleaf and WPS are trademarks of Interleaf, Inc.

Preface

This manual describes the binder (BIND), the librarian (LBR), and installed libraries (INLIB, etc.). We've organized this manual as follows:

Chapter 1	Explains the program development process on a DOMAIN system. Here, we introduce the binder, the librarian, installed libraries, and the loader.
Chapter 2	Describes the syntax of the binder utility.
Chapter 3	Describes the syntax of the librarian utility.
Chapter 4	Details installed libraries and explains how to use the INLIB utility.
Appendix A	Lists some of the binder warning and error messages, and provides possible remedies.
Appendix B	Lists some of the librarian warning and error messages, and provides possible remedies.
Appendix C	Explains all the section attributes.
Appendix D	Provides a tutorial session using various common binder and librarian options. The programs used in the session can be found in an on-line directory.

Related Manuals

The following language manuals should be used in conjunction with this manual:

- The *DOMAIN Pascal Language Reference* (000792).
- The *DOMAIN FORTRAN Language Reference* (000530).
- The *DOMAIN C Language Reference* (002093).

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. Refer to the CRUCR (CREATE_USER_CHANGE_REQUEST) Shell command description. You can view the same description on-line by typing:

$ HELP CRUCR <RETURN>

For your documentation comments, we've included a Reader's Response form at the back of each manual.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

boldface	Bold, uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally. Letters in uppercase boldface must be used, but letters in lowercase boldface are optional. For instance, consider **SIGnal**. Since the word is boldfaced, it is mandatory. The arrangement of uppercase and lowercase letters indicates that the word can be abbreviated to SIG.
nonboldface	Words that are neither boldfaced, nor italicized indicate a part of the expression that you must supply, but you do not supply it literally. For instance, consider pathname. Here, you must enter an argument. You would not enter the word "pathname," you would enter a pathname instead.
italicized	Italicized words are optional arguments.
`output`	Typewriter font words in command examples represent literal system output or pathnames.
`color`	Words printed in colored ink represent sample user input.
<RETURN>	A word enclosed in angle brackets indicates a key on the keyboard. For example, <RETURN> symbolizes the RETURN key.
( )	In examples, parentheses enclose comments.
...	Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.
⋮	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Summary of Technical Changes

We last revised this manual for AEGIS SR9.0. Since then, we've reorganized the manual and improved its format. The LBR and INLIB utilities have not changed technically since SR9.0; however, we've made the following changes to the binder since then:

- We've added a new –INLIB/–NOINLIB binder option. The –INLIB option is an alternative to the INLIB utility.
- We've added a new –ENTRY binder option which allows you to specify a nondefault program starting address.
- We've added a new –ALLKEEPMARK option which preserves the marks on global symbols.
- We've added a –MERGEBSS binder option which merges all C global symbol sections into one section named BSS$.

- If you specify an unrecognized option in interactive mode, the binder treats it as a warning, rather than an error.

- The –BINARY option no longer allows the output binary file to begin with a hyphen (–).

- We increased the maximum number of sections per module from 2048 to 3072. This primarily benefits C programmers (the C compiler creates a section for each global variable).

- When the binder cannot find or read a binary input file, it no longer prints out two different pathnames in the warning or error message.

We've used changed bars to mark technical changes to the utilities since SR9.0.

Contents

Illustrations

Tables

Chapter 1

Program Development on the DOMAIN System

This chapter is a general introduction to program development on your DOMAIN system. Here, we describe the role each that of the following programs and utilities plays in program development:

- The Compilers
- The Binder
- The Librarian
- Installed Libraries
- The Loader

If you want to know how to operate the compilers, see the appropriate language manuals (listed in the Preface). To learn how to use the binder and the librarian, see Chapters 2 and 3, respectively, of this manual. In Chapter 4 we detail installed libraries and the use of the INLIB utility. We describe the loader in Chapter 1.

1.1 Program Development -- An Overview

Describing exactly how you should develop a DOMAIN FORTRAN, Pascal, or C program is not a trivial task, as there are many program development paths for you to choose from. However, the start of program development is always the writing of source code. When writing source code, you can put it all into one file, or you can spread it out over several files. After writing the source code, you must compile each file separately. The compiler produces an **object file**, which is a file (usually with the filename extension .BIN) that contains one or more object modules. (An object file created by the C or Pascal compiler always contains one object module; an object file created by the FORTRAN compiler can contain one or more object modules.) An **object module** contains machine language code and data in a form that can be used by the binder, the librarian, INLIB, or the loader. You can do the following with an object file:

- In some cases, you can execute or debug the object file directly. However, in other cases, you won't be able to execute or debug an object file until you use the binder to bind it with other object files. (Chapter 2 specifies the cases in which you must bind.) The binder creates an output object file which can be used like any other object file. In fact, you can use the output object file as an input file to a subsequent binder command.

- You can use the librarian to create a library file out of one or more object files. You can use a library file as input to the binder or as input to another librarian command.
- You can use INLIB to install an object file as an installed library. (We explain what an installed library is later in this chapter.)

Figure 1-1 illustrates the interaction of the various utilities. In the upcoming sections, we take a closer look at each of the program development components on your DOMAIN system.

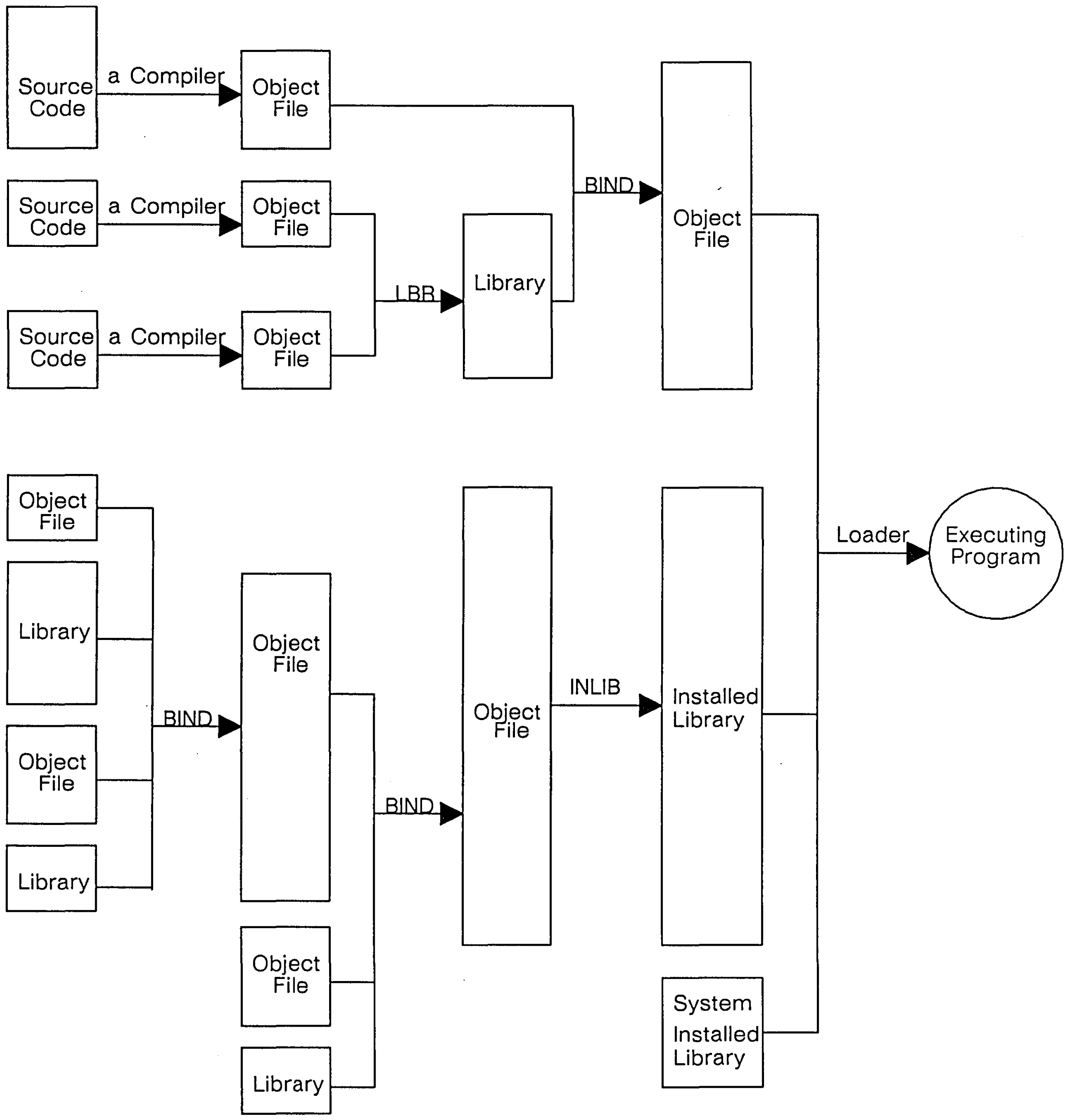

Figure 1-1. The Interaction of Program Development Utilities

1.1.1 The DSEE System

In addition to the traditional programming development scheme shown in Figure 1–1, you can also use the DOMAIN Software Engineering Environment (DSEE) system to develop DOMAIN programs. The DSEE package is a support environment for software development. DSEE helps engineers develop, manage, and maintain software projects; it is especially useful for large–scale projects involving a number of modules and developers. You can use DSEE for:

- Source code and documentation storage and control
- Audit trails
- Release and Engineering Change Order (ECO) control
- Project histories
- Dependency tracking
- System building

In this chapter, we describe a traditional program development cycle; the DSEE product provides some sophisticated enhancements to this cycle. For information on the optional DSEE product, see the *DOMAIN Software Engineering Environment (DSEE) Reference*.

1.2 The Compilers

If you've written source code without errors, a compiler will create an object file. In this section, we concentrate on two aspects of object files that are particularly relevant to program development.

First consider global symbols and external references. A **global symbol** is a definition that can be accessed by code in another object file. A global symbol can be a subroutine name or a common block name in FORTRAN. Also, a global symbol can be the name of an external procedure or external variable in Pascal or C. An **external reference** is an attempt by a piece of code in one object file to access a global symbol defined in another object file. A compiler cannot resolve external references because it does not read anything other than the source code it is compiling. Therefore, a compiler simply flags the external reference and waits for the binder or loader to match it with a global symbol.

The second compiler concept worth noting is the **section**. A section is a named area of code or data that shares a set of **attributes**. An attribute is an instruction to the binder, the loader, or INLIB to create or execute an object file in a particular way. For example, a section can be made –READONLY to protect its data. You can control some attributes through your source code –– some when you compile, some when you bind, and some not at all. (See Appendix C for a complete list of attributes.) When the compiler creates an object module, it notes which section each piece of code or data belongs to.

1.3 The Binder

The **binder** combines one or more input object modules to form one output object file. For example, the following bind command reads in the input object modules `compute_tax.bin` and `luxury_tax.bin` to create one output object file named `taxes`:

```
$ bind compute_tax.bin luxury_tax.bin -binary taxes
```

An important feature of the binder is that it resolves external references by matching them with their global symbol counterparts. For example, in the previous example, suppose that `compute_tax.bin` makes an external reference to a procedure called `luxury_tax` and that `luxury_tax.bin` defines the procedure `luxury_tax` as a global symbol. In this case, the binder matches the external reference with the global symbol so that in `taxes`, the code that was trying to call `luxury_tax` will be able to access it.

The binder combines only the object files you explicitly name on the binder command line. Therefore, if you have experience with other operating systems, you may be wondering "How can the program execute without operating system routines and language routines?" The answer is that DOMAIN programs do require such routines, but your program will access them at runtime instead of at bind time. At runtime, an internal utility called the loader matches your program's requests for service routines with the service routines available in the installed libraries.

Although the binder does not load routines from the installed libraries, it does look at a list of the global symbols defined by the installed libraries. The binder checks this list to see if external references not resolved within the input object modules can be resolved by a symbol in the installed libraries. If so, the binder concludes that the unresolved external reference will be resolved at runtime, and therefore the binder does not issue a warning. However, if an unresolved external reference cannot be resolved by a global symbol in the list, then the binder issues a warning.

For example, suppose one of your input object modules makes an external reference to the symbol `ios_$get` and that none of the input object modules define a global symbol `ios_$get`. Therefore, the binder checks to see whether `ios_$get` is defined as a global symbol in an installed library. If it is, the binder assumes that the loader can resolve `ios_$get` at runtime, and, therefore, the binder does not issue a warning. If `ios_$get` is not defined as a global symbol in an installed library, then the binder knows that the loader will not be able to resolve `ios_$get` at runtime, and, therefore, the binder issues a warning.

1.4 The Librarian and Library Files

The **librarian** is the utility that creates, edits, and describes library files. A **library file** is a special file created by the librarian, consisting of one or more object modules collected together for easy access by the binder. Using the librarian, you can delete, extract, or replace object modules stored in a library file. Typically, you store object modules in a library file so that the binder can load a subset of them.

You can use a library file as input to another librarian command or as input to the binder. You cannot use a library file as input to INLIB, and you cannot use a library file as input to the loader. That is, you cannot execute a library file.

The binder accepts both library files and object files as input. The binder *unconditionally* loads *all* the object modules stored in object files. However, the binder loads an object module stored in a library file only if at least one of the following conditions is true:

- You explicitly specify that object module for loading with the binder switch -INCLUDE.
- The object module satisfies an unresolved external reference.

Basically, a library file makes binding easier by allowing you to store a number of object modules inside one file. Therefore, you can simply put the name of one library file on the binder command line rather than listing hundreds of individual object files. Furthermore, because the binder conditionally loads object modules in a library file, your program won't contain unnecessary code.

NOTE: Do not confuse a "library file" with an "installed library." Remember that only the librarian can create a library file.

1.5 Installed Libraries

An **installed library** is a set of one or more object modules stored in a way that permits access by the loader only. The loader uses the routines in installed libraries to resolve outstanding external references made by running programs. In other words, an installed library contains code and data that your program can optionally execute at runtime.

For example, suppose global symbol `fed_tax` is defined in an installed library. Therefore, your object file can make an external reference to `fed_tax` without resolving it at compile time or bind time. At runtime, the loader will resolve the external reference from your program with the global symbol in the installed library.

Installed libraries are an often–overlooked feature of the DOMAIN operating system. They provide an alternative to binding required libraries to each program that uses them. Installed libraries result in much smaller object module sizes on disk because they eliminate the replication of bound library copies. They also allow for more efficient use of physical memory when two or more programs executing concurrently require the same library. That's because the programs all share the same physical copy of the installed library in memory. Finally, the use of installed libraries makes it much easier to update the software on a node. When a new version of an installed library is available, the user need only copy it into those programs. This is vastly simpler than having to locate all bound images that use the library and rebind them to use the new version.

There are five types of installed libraries:

- User–defined installed libraries
- Object files installed with the –INLIB binder option
- System–defined installed libraries
- System–defined global libraries
- User–defined global library

We detail all five types in Chapter 4.

1.6 The Loader

The loader is the DOMAIN utility that oversees the execution of all user programs. When you enter the name of a file to be executed, the loader prepares main memory for executing the program. A very important function of the loader is to resolve outstanding external references in your program with global symbols defined in the installed libraries. Thus, you might think of the loader as sort of an internal binder utility.

As we noted before, there are no loader options or ways to talk to the loader. It is automatically invoked whenever you attempt to execute a program.

1.7 Limits and Restrictions

Be aware of the following limits when you develop programs:

- The combined number of sections and marked global symbols in an object module cannot be greater than 3,072.
- The combined number of sections and marked global symbols in an installed library cannot be greater than 3,072.
- Only the first 32 characters of a global variable or section name are significant.

The binder can build a program of any size. That is, the output object module can be any size. However, the ultimate restriction on the size of the program is based on the available disk space and the available address space.

Chapter 2

How to Use the DOMAIN Binder

This chapter explains when and how to use the DOMAIN binder utility. You should also refer to the appropriate language manual for information about source code and its effect on binding.

2.1 When to Use the Binder

You must use the binder if any of the following conditions are true:

- Your FORTRAN, Pascal, or C program consists of more than one source file.
- Your FORTRAN, Pascal, or C program requires an object module from a library file.
- Your one–file FORTRAN program contains more than one program unit.

In general, binding is unnecessary when your program consists of only one file. However, if that program requires any object modules from a library file, then it will be necessary to bind. A program does not require binding if it only requires object modules from installed libraries.

2.2 How to Invoke the Binder

To invoke the binder, type a command line of the following format:

```
$ BIND pathname1 ... pathnameN  option1 ... optionN
```

In other words, the command line simply consists of the word BIND, one or more pathnames, and zero or more options. Note that the command format is somewhat misleading in that some options must precede pathnames but others must follow the pathnames.

The binder uses the object modules stored in **pathname** to create an executable object. A pathname must be the name of a valid object file or valid library file. (A compiler creates a valid object file, and the librarian creates a valid library file.) You can use wildcards in pathnames. The binder automatically loads all object modules stored in object files, but conditionally loads the object modules stored in libraries. For details about how the binder loads the object modules in library files, see Chapter 3.

Options, detailed in the next section, modify the binder's actions. Of all the binder's options, -BINARY is the most important. You must use this option to get an executable object. For example, the following command line combines object files plot_data.bin and drawings.bin into the executable object file plot_data:

```
$ bind plot_data.bin drawings.bin  -binary plot_data
```

The binder processes arguments sequentially. However, the order of binary files is not important. That is, you do not have to list the main program first followed by the subroutines. Most options apply to files you specify later in the command string, but have no effect on the files previously specified. This feature allows you to turn options on and off, from one file to the next.

2.2.1 Multilevel Binding

Multilevel binding means binding to create an output object module, and then using that output object module as an input object module to a second binder command line.

For example, suppose you issue the following command:

```
$ bind a.bin b.bin -binary lev1
```

Single-level binding means that you do not use lev1 as an input object module in another binder command line. Multilevel binding means that you use lev1 as an input object module to another binder command line, as in the following:

```
$ bind lev1 c.bin d.bin -binary lev2
```

Multilevel binding is particularly useful when developing a program that consists of many, many object files. For instance, suppose your program consists of 100 object files, but that you want to fix bugs in only one of those files. If multilevel binding did not exist, you would have to rebind all 100 object files each time you recompiled. A more efficient scheme would be to bind the 99 unchanging object files once, and then rebind this object file to the file that keeps getting recompiled.

2.2.2 Spreading a Binder Command Over Several Lines

If you want to spread a binder command over more than one line, then you must either:

- Put a hyphen (-) at the end of the first line.
- Enter the command BIND (and nothing else) as the first line.

To signal the end of a spread binder command, you must either:

- Put -END at the end of the command.
- Leave the final line blank.

For example, the following three binder command lines are equivalent. All three create an executable object (in file roll_em) out of three object files:

```
$ BIND lights.bin -
* cameras.bin action.bin
* -BINARY roll_em -END
```

or

```
$ BIND lights.bin -
* cameras.bin action.bin
* -BINARY roll_em
*
```

or

```
$ BIND
* lights.bin  cameras.bin action.bin
* -BINARY roll_em
*
```

2.2.3 Comments

You can add comments to bind arguments. The binder ignores the text of comments. Delimit them with braces { }, as shown below:

```
$ BIND sio.bin -
* {This is a comment.}
* rw.bin
* math.bin {This is another comment}
* -binary sio -end
```

If you forget to terminate a comment with a closing brace }, <RETURN> will terminate the comment. Therefore, if you try to span a comment over two lines without starting the comments on both lines with beginning braces {, the binder will interpret your comment as a list of pathnames. For example, compare the right and wrong ways to specify a multiline comment.

```
$ bind apples.bin -
* oranges.bin {This is a }
*             {good comment}
* lemons.bin
* pears.bin {This is a
*            bad comment}
```

2.2.4 Errors

If a problem occurs during binding, the binder displays a message in standard error output. The message indicates the nature and severity of the problem. The binder issues two kinds of messages: warning-level and error-level. Warning-level messages indicate conditions that do not prevent the binder from producing an output file. However, warning-level messages may mean that the file's contents are not what you expect. Error-level messages are fatal conditions that prevent the binder from producing an output file.

Appendix A lists all binder error and warning messages and gives an explanation of the likely cause of each problem.

If a binder command line generates an error, then the binder does the following. First, the binder looks in the appropriate directory for a file with the same name as the file it would have created had binding succeeded. Then, if this file exists, binder adds the .bak extension to its name. If this file doesn't exist, then the binder takes no action. If you later rebind successfully, the binder deletes the .bak file when creating the executable object file. For example, consider the following series of bind command lines.

First, binder creates filename q:

```
$ bind geoshapes.bin math1.bin math2.bin math3.bin -binary q
All Globals are resolved.
```

Next, due to a binder error, binder changes the name of q to q.bak and does not create a new q:

```
$ bind geoshapes.bin math1.bin math2.bin math3.bin -ruff -binary q
?(bind) Error: Unknown Command Ignored
```

Finally, we rebind correctly causing the binder to delete `q.bak` and create a new `q`:

```
$ bind geoshapes.bin math1.bin math2.bin math3.bin -binary q
All Globals are resolved.
```

Undefined Global Symbols Errors

If you forget to type the pathname of an object module you want to bind, the binder may report undefined global symbols upon completion. This in itself is not a fatal error. If this occurs, you need not rebind all modules. Just bind the resulting output module with the module you previously omitted. There is no limit to the number of times a module can be bound. See the "Multilevel Binding" section earlier in this chapter for details.

2.3 Binder Option Summary

The bulk of this chapter is devoted to descriptions of all the binder options. We begin with a list of all the options and their syntax. A few notes on typographical conventions are now in order. Consider, for example, the following entry:

-Binary pathname

The fact that **-Binary** is in boldface tells you that this part of the option is to be entered literally; however, the lowercase letters **inary** are optional. In other words, you can specify this option as -Binary or as -B. The fact that pathname is not boldfaced tells you that this part of the option is to be entered nonliterally. In other words, do not enter the word "pathname," enter a pathname instead.

Option and Syntax	Purpose
-ALIGN section_name **LONG**	Aligns the named section on a 32-bit boundary at runtime.
-ALIGN section_name **QUAD**	Aligns the named section on a 64-bit boundary at runtime.
-ALIGN section_name **PAGE**	Aligns the named section on a 8,192-bit boundary at runtime.
-ALLKEEPMARK	Preserves all marks.
-ALLMARK	Marks all global symbols in the input object files that appear after the option on the bind command line.
-ALLRESolved	Signals a shell severity level of "error" if there are unresolved global symbols at the end of a bind command. Useful in controlling Shell scripts.
-ALLUNMARK	Unmarks all global symbols in the input object files that appear after the option on the bind command line. (DEFAULT)
-BDIR directory_name	Adds a pathname to the list of directories that the binder searches in for input object files.
-Binary pathname	Creates an output object module and stores it at pathname.
-END	Signifies the end of a command that is spread over several lines.
-ENTRY global_symbol	Specifies a nondefault start address.

-EXACTCASE	Makes the binder case-sensitive to all variable names and section names.
-GLObals	Writes currently defined global symbols to error output.
-Help	Prints this list of commands.
-INCLude module_name	Unconditionally loads the named object module from a library file into the output object file.
-INCLude -ALL	Unconditionally loads all object modules from a library into the output object file.
-INLIB pathname	Specifies that the object file(s) in pathname are to be "installed" when the output object file is invoked. (This is an alternative to using the -INLIB utility.)
-LOCALSEARCH	Forces the binder to make another search through a library file if the previous search loaded an object module containing an unresolved external reference.
-LOOKSection section_name	Makes the named section available for sharing with a public section in an installed library.
-LOOKSection -ALL	Makes all subsequent sections available for sharing with their counterpart public sections in an installed library.
-MAKers	Lists the version numbers of the compilers, binders, etc., that were used to create the input object files.
-MAP	Writes a complete binder map to standard output.
-MARK global_symbol	Marks the specified global symbol.
-MARK -ALL	Same as -ALLMARK.
-MARKSection section_name	Makes section_name public. Affects only those object files that are destined to be installed as an installed library.
-MARKSection -ALL	Makes all subsequent sections public. Affects only those object files that are destined to be installed as an installed library.
-MERGEbss	Merges all sections corresponding to C global variables into a single section named BSS$.
-MESsages	Produces informational messages at the end of a bind command. (DEFAULT)
-MODule new_name	Changes the name of the output object module from the default (the first input object module loaded) to new_name.
-MSGS	Same as -MESSAGES. (DEFAULT)
-MULTIRES	Reports errors if multiple resolutions of the same external symbol exist in object module libraries.
-NMSGS	Same as -NOMESSAGES.
-NOEXACTCASE	Sets the binder to ignore case differences on names. (DEFAULT)

-NOINLIB pathname	Specifies that the object file(s) in pathname are no longer to be "installed" when the program is invoked.
-NOLOOKSection section_name	Makes the named section unavailable for sharing.
-NOLOOKSection -ALL	Makes all subsequent data sections unavailable for sharing.
-NOMARKSection section_name	Makes section_name private.
-NOMARKSection -ALL	Makes all subsequent sections private.
-NOMESsages	Suppresses informational messages at the end of a bind command.
-NoMULTIRES	Omits error reporting for mulitple resolutions in object module libraries. (DEFAULT)
-NoUNDefined	Suppresses the listing of undefined globals.
-Quit	Exits from the binder without finishing.
-READONLYsection section_name	Changes the read/write attribute of section_name to read-only.
-SECtions	Displays a section map.
-SET_VERsion number.number	Sets the program version in the map to the specified number.
-SORTLocation	Sorts global symbols numerically (by position).
-SORTNames	Sorts globals symbols alphabetically (by name). (DEFAULT)
-SYStem	Makes system globals visible.
-SYSTYPE	Builds a shared resource record into the bound output module. Specify valid system names, such as sys3, sys5, bsd4.1, or bsd4.2. This option overrides all system information from the object modules.
-UNDefined	Suppresses a listing of unresolved external symbols present at the end of a bind command line.
-UNMARK global_symbol	Removes a mark from the specified global symbol.
-UNMARK -ALL	Same as -ALLUNMARK.
-UNMARKSection name	Makes section_name private. Affects only those object files that are destined to be installed as an installed library.
-UNMARKSection -ALL	Makes all subsequent sections private. Affects only those object files destined to be installed as an installed library.
-XREF	Displays a listing of cross-references.
- (hyphen)	Tells the binder that more input will following on the next line.

2.4 Detailed Descriptions of Each Binder Option

The remainder of this chapter details each binder option.

-ALIGN -- Aligns the specified section on a boundary (which may improve execution speed.)

FORMAT

		You must specify one of the following three:
-ALIGN	section_name	LONG QUAD PAGE

ARGUMENTS

section_name	The name of the section you want to align.

The argument after section_name must be one of the following three:

LONG	to align the section on a long (32-bit) boundary.
QUAD	to align the section on a quad (64-bit) boundary.
PAGE	to align the section on a page (8,192-bit) boundary.

DESCRIPTION

The -ALIGN option directs the loader to align the section you specify on a 32-bit (LONG), a 64-bit (QUAD), or an 8192-bit (PAGE) boundary. The default is LONG. Alignment on a LONG boundary boosts a program's performance on certain nodes, such as the DN460, DN660 and DSP160.

You cannot place an -align option before the section is defined by one of the object modules. That is, if you specify -align section_name, but section_name has not been defined yet, then the binder will report an error.

EXAMPLES

Suppose that we wanted to align a section named big on a page boundary. To do so, we'd issue a command like the following:

```
$ bind one.bin two.bin  -align big page  -binary my_program
```

Don't forget that, by default, global variables in C programs are named sections.

-ALLRESOLVED -- Causes the binder to generate a severity level of "ERROR" if it encounters any unresolved global symbols.

FORMAT

-ALLRESolved

DESCRIPTION

This option affects the severity level issued by the binder if it encounters any unresolved global symbols. Readers unfamiliar with the concept of severity level should read about it in the *Getting Started With Your DOMAIN System* manual.

If you do not specify the -ALLRESOLVED option, then unresolved global symbols do not affect the severity level issued by the binder. If you do specify the -ALLRESOLVED option and the bind command contains unresolved global symbols, then the binder will generate a severity level of ERROR.

EXAMPLE

Consider the following Shell script:

```
# Without -ALLRESOLVED
bind geoshapes.bin math1.bin math2.bin -binary my_program
args "This is line 3."
```

Here's what happens when we execute the script:

```
$ script
Undefined Globals:

   circle                             First referenced in GEOSHAPES.BIN

This is line 3
```

Now consider what happens when the bind command line contains an -ALLRESOLVED option:

```
# With -ALLRESOLVED
bind geoshapes.bin math1.bin math2.bin -allresolved -binary my_program
args "This is line 3."
```

Executing this script results in an error-level severity (thus forcing an immediate exit from the script):

```
$ script
Undefined Globals:

   circle                             First referenced in GEOSHAPES.BIN

?(bind) Error: Not all globals were resolved

1 Error.
```

-BDIR -- Adds a pathname to the list of directories the binder searches for input object files.

FORMAT

-BDIR directory_pathname

ARGUMENTS

directory_pathname The pathname of a directory that you want the binder to search.

DESCRIPTION

Use the -BDIR option to tell the binder to search for input object files in a directory other than the working directory. The -BDIR option only affects input object files having relative pathnames; it does not affect those having absolute pathnames.

An absolute pathname begins with a slash (/), double slash (//), tilde (~), or period (.); for example:

```
//tesich/breaking/away.bin
/reiner/stand/by/me
~truffaut/wild/child.bin
.kurys
```

A relative pathname is any pathname that does not begin with a slash (/), double slash (//), tilde (~), or period (.). For example, any pathname that begins with a name is a relative pathname. Note that a relative pathname specifies a file in a way that is relative to your working directory.

Now that we've distinguished between absolute and relative pathnames we can describe how -BDIR works. If you do not specify the -BDIR option, the binder looks for each input object file using only the file pathname you specify. If the input file cannot be found under the pathname, the binder reports a warning or error as appropriate. If, on the other hand, you specify the -BDIR option, then when the binder cannot find an input file having a relative pathname, it goes on to search for that file in the -BDIR directory. To do so, it prefixes the -BDIR directory's pathname, separated by a slash (/), to the relative pathname you supplied for the input file.

Notice that the -BDIR option can only take one directory_pathname as an argument. Therefore, if you want the binder to search multiple directorys, you must specify multiple -BDIR options. The binder will always search the working directory first, then it will search the -BDIR directories in the order that they appear on the command line.

Bear in mind that the -BDIR option must precede on the command line the input object files that you want it to affect. In attempting to locate a given input file, the binder does not take into account any -BDIR options that follow that input file's pathname on the command line.

EXAMPLES

Suppose that you are developing a program, and that you keep copies of all of the program's consituent object files (a.bin, b.bin, and c.bin) in the directory //spielberg/develop/bins. When you want to work on some piece of the program, you place copies of the appropriate source files in your working directory, make your changes, and then compile them, directing the compiler to output the new object files into your working directory. (You don't want them output directly into the central repository of object files because you haven't debugged them yet. Once you have done so, you will copy them into the central repository.)

You want a Shell script that you can run to bind the program together, one that will prefer any object files in your working diretory over the copies in the central repository, taking the remaining object files from the central repository. You do not want to have to change this Shell script each time you begin work on a different piece of the system. That is, you do not want to have to say ahead of time what object files will be in your working directory and what ones will not.

The Shell script might contain the following binder command line:

```
$ bind -bdir //spielberg/dev/progs a.bin b.bin c.bin -binary my_program
```

This command causes the binder to pick up any or all of the input files `a.bin, b.bin, c.bin` from your working directory, depending on which are present there. For any that are not present in your working directory, the binder gets them from `//spielberg/develop/bins`. Therefore this command will bind together a program containing your changes no matter what piece of the system you happen to be working on.

-BINARY -- Creates an executable object file.

FORMAT

-Binary pathname

ARGUMENTS

pathname	The pathname of the binary file you are trying to create.

DESCRIPTION

Virtually all binder command lines contain this option. It causes the binder to create an executable object file and store it at a specified pathname.

To avoid confusion, try not to choose a pathname of an input object file as the argument to -BINARY.

You can specify the -BINARY option anywhere in the binder command line, but you may not specify it more than once. That is, the binder can only create one object file per command line.

Remember, if you don't use -BINARY, the binder won't create an executable object; however, the binder will report errors and produce maps.

EXAMPLE

The following two commands are equivalent. Each produces an output object file named my_program.

```
$ bind one.bin two.bin -b my_program
$ bind one.bin two.bin -binary my_program
```

-END -- Terminates a bind command that spans multiple lines.

FORMAT

-END

DESCRIPTION

Use the -END option to mark the end of a binder command that extends over two or more lines.

EXAMPLES

To end a binder command line that spans multiple lines, you can either use the -END option or leave the final line blank. For example, first we use the -END option:

```
$ bind lights.bin -
* cameras.bin action.bin
* -binary roll_em -end
$
```

Here we leave the final line blank:

```
$ bind lights.bin -
* cameras.bin action.bin
* -binary roll_em
*
$
```

-ENTRY -- Specifies a nondefault program start address.

FORMAT

-ENTRY global_name

ARGUMENTS

global_name	The name of a global symbol previously defined in the command line. Typically, it is the name of a routine.

DESCRIPTION

By default, the output object module created by the binder specifies a start address corresponding to the first executable instruction in the object module's main program ("main()" in C, "program" in Pascal or FORTRAN), if any. The -ENTRY option allows you to specify an alternative start address.

EXAMPLES

The following command does not contain an -ENTRY option; therefore, the binder sets the start address equal to the first executable instruction in the main routine:

```
$ bind geometry.bin circles.bin  -binary geom
```

Suppose though that we did not want the default start address. Instead, we wanted the program start address to be the first executable instruction in a routine named `pie` (defined somewhere in `geometry.bin` or `circles.bin`). To accomplish this, we would issue the following command:

```
$ bind geometry.bin circles.bin  -binary geom  -entry pie
```

-EXACTCASE -NOEXACTCASE -- Makes the binder case-sensitive or case-insensitive to the arguments of binder options.

FORMAT

-EXACTCASE
-NOEXACTCASE

DESCRIPTION

The -EXACTCASE option causes the binder to distinguish between uppercase and lowercase letters in names you use as arguments for such bind options as -INCLUDE, -MARKSECTION, and -ALIGN. The default is -NOEXACTCASE. Thus, normally the binder ignores case differences, treating uppercase and lowercase letters identically.

The -EXACTCASE option is primarily intended for object modules in the C language, since the C compiler produces case-sensitive names. The -EXACTCASE option has no user-visible impact on FORTRAN or Pascal object modules.

EXAMPLES

Consider a C global variable (and therefore a section) named big. If we do not use the -EXACTCASE option, then the binder is case-insensitive, so the following two commands produce exactly the same results:

```
$ bind a.bin b.bin -binary my_program   -align big page
$ bind a.bin b.bin -binary my_program   -align BIG page
```

However, if we use the -EXACTCASE option, then the first command produces the correct results and the second command causes errors:

```
$ bind -exactcase a.bin b.bin -binary my_program  -align big page (okay)
$ bind -exactcase a.bin b.bin -binary my_program  -align BIG page (error)
```

-GLOBALS -- Causes the binder to display the current global map.

FORMAT

-GLObals

DESCRIPTION

The -GLOBALS option causes the binder to display the current global map. The global map contains the name and position (as an offset from the beginning of a section) of all global symbols defined until that point.

If you use the -MAP option, the binder displays the current global map as part of a larger listing. (So placing -GLOBALS and -MAP adjacently on the same command line produces redundant information).

If the message "`No defined Globals`" appears in the global map, it means that none of the input object modules defined a global symbol.

Note that the binder prints the section map based on the object modules preceding -SECTIONS on the command line. In other words, the section map that the binder produces depends on the position within the command line of -SECTIONS.

EXAMPLE

```
$ bind a.bin b.bin c.bin  -mark nick  -binary abc  -globals
Global Map:
  Offset  In Section  Name
 00000018       2    <apollo_c_startup>
 000000D0       2    b
 00000000       4    big
 000000F8       2    catch
 0000002C       2    main
 00000000       6    nick                       marked
 00000000       7    rachel
 00000000       5    str
All Globals are resolved.
```

The Global Map Explained

The Global Map describes each global symbol defined by the input object modules. The global map lists each global symbol's name and position. The position is described as a hexadecimal offset from the beginning of a particular section. For example, symbol `catch` is defined $F8_{16}$ bytes past the beginning of section 2. All symbols with the same section number are stored in the same section.

The word "marked" indicates that a particular symbol is marked. See the "-MARK" listing later in this encyclopedia for an explanation of "marked" and "unmarked."

The binder prints the message "`All Globals are resolved`" in the following cases:

- You used the -SYSTEM option but your program makes no references to a symbol in an installed library.

- You did not use the –SYSTEM option, and all external references made by the input object modules can be resolved by other input object modules or by installed libraries.

If neither is the case, the listing would show all the unresolved external references beneath the heading "Undefined Globals". Next to each symbol name, the listing shows the pathname of the file that the symbol was "First referenced in". That is, if one or more input object files make an external reference that the binder cannot resolve, the listing shows the pathname of the object file that referred to it first.

–INCLUDE -- Forces the binder to load one or more object modules from a library file into the output object module.

FORMAT

		You must select one of these two:
library_pathname	**–INCLude**	object_module_name –ALL

ARGUMENTS

library_pathname The last pathname that precedes –INCLUDE must be a library file. (See Chapter 3 for a definition of library files.) –INCLUDE affects this library file only. If no library file precedes –INCLUDE or if there is some other pathname between the library file and –INCLUDE, then the binder issues an error message.

You must select one of the following two arguments following the keyword –INCLUDE:

object_module_name The name of one object module stored in library_pathname. (The binder issues an error message if object_module_name is not stored in the library.) The binder will automatically load this object module into the output object file. You cannot use a wildcard in object_module_name.

–ALL The keyword –ALL causes the binder to automatically load every object module from library_pathname into the output object file.

DESCRIPTION

By default, the binder loads an object module from a library file if it resolves an external reference. If you specify –INCLUDE, then one or more object modules from the library file will be loaded whether or not they resolve an external reference.

EXAMPLES

For example, suppose that `math.lib` is a library file consisting of object modules `geom`, `trig`, and `calculus`. If you want to ensure that the binder loads `trig` into the output object module (waves), then you could issue the following command:

```
$ bind a.bin b.bin math.lib -include trig -binary waves
```

If you had wanted to force–load both `geom` and `trig`, then you would have issued the following command:

```
$ bind a.bin b.bin math.lib -include geom -include trig  -binary waves
```

If you want to ensure that the binder loads all three object modules in `math.lib`, then you should issue the following command:

```
$ bind a.bin b.bin math.lib -include -all   -binary waves
```

You need to use –INCLUDE to *ensure* that the binder loads an object module from a library file into the output object module. If you don't use –INCLUDE to load an object module, the binder will still load it if it satisfies an unresolved external symbol. See Chapter 3 for more information on library files.

-INLIB, -NOINLIB -- Causes an object module to be installed when the output object file is executed.

FORMAT

-INLIB pathname
-NOINLIB pathname

ARGUMENTS

pathname	The pathname of the object module you want to install. The pathname must have been produced by a compiler or the binder, but not by the librarian. The specified pathname is the one that will be used by the loader to locate the installed library object module at execution time. The binder makes no attempt to use this pathname at bind time. The binder merely writes the name into the output object module for later use by the loader.

DESCRIPTION

Use –INLIB as an alternative to the INLIB shell command. The –INLIB binder option makes code available to an executing program without actually binding the code into the output object file. (For some background information on installed libraries, see Chapter 1; for a detailed comparison of various kinds of installed libraries, see Chapter 4.)

Note that the new –INLIB binder option does not replace the DOMAIN INLIB utility or the USERLIB.PRIVATE facility. It merely provides an alternate way of installing code on a per–program basis, rather than on a per–shell or per–login session basis. (See Chapter 4 for complete details about the other kinds of installed libraries.)

At bind time, the binder does not read from the object file specified in an –INLIB option. Therefore, the binder will list as "unresolved" any references to global symbols in this object file. Rest assured, though, that the loader will resolve these references at execution time. If you don't want these references listed as "unresolved," then use the INLIB utility to install the appropriate object files in the process prior to invoking the binder. The marked global symbols in the installed object file will then be known to the binder, just as the globals defined by system libraries are known to the binder. Although the binder will still not actually resolve references to those globals at bind time, it will not list them as unresolved.

The map produced by the binder's –MAP option includes information about any installed libraries required by the object module.

-NOINLIB

The –NOINLIB option undoes the actions of the –INLIB option; that is, it makes the code unavailable to the executing program. You would probably only use this option in one of the later steps of a multilevel bind.

The –NOINLIB option applies only to installed libraries specified *earlier* in the command line, either directly via the –INLIB option or indirectly by an input object module. The binder issues a warning if it encounters a –NOINLIB option specifying an installed library pathname that is not (yet) known to the binder.

Installing More Than One Object File

If you want more than one object file to be installed when you invoke the program, then you must specify more than one –INLIB option on the bind command line. For example, if you want to install tthe object files /lib/highlib and /lib/lowlib whenever my_program is invoked, you would issue a bind command like the following:

```
$ bind a.bin b.bin -inlib /lib/highlib -inlib /lib/lowlib  -b my_program
```

If you put multiple -INLIB options on the same bind command line, the system will usually install the objects in the same order that you specified them. However, we do not guarantee it; therefore, you should not depend on it.

Instead of using multiple -INLIB options on the same command line, you can which set up a chain of dependencies with -INLIB. For example, suppose you are building my_program which needs to call /lib/highlib, but /lib/highlib in turn needs to call installed object file /lib/lowlib. Here is what you do:

```
$ bind c.bin d.bin -inlib /lib/lowlib  -b /lib/highlib
$ bind a.bin b.bin -inlib /lib/highlib -b my_program
```

Therefore, when you invoke my_program, the loader installs lowlib and then installs highlib.

Actually, the order you install libraries in is immaterial in most cases. Interlibrary dependencies that result from procedure calls do not necessitate that the libraries be installed in any particular order. However, inter–library dependencies that result from data references can impose ordering constraints. Thus, if /lib/highlib refers to named global data defined in /lib/lowlib, then /lib/lowlib must be installed prior to /lib/highlib.

How The Loader Locates Pathnames

In order to load a program that requires installed libraries, the system loader must determine whether those libraries are already installed in the process, and if not, install them. To determine whether a library with a given pathname is already installed in the process, the loader first identifies the file system object that (currently) has that pathname. The loader then determines whether *exactly that object* has been installed in the process. Thus, the determination is *not* based solely on a comparison of pathnames.

Suppose, for example, that /lib/lowlib was installed earlier in the process, but since that time its name has been changed to /lib/baselib. If the loader is then told to load a program that requires the installed library /lib/baselib, it will detect that the file system object (currently) having the pathname /lib/baselib is in fact already installed in the process. It can make this determination even though that object has a different pathname now than it did when it was first installed in the process.

As another example, suppose that /lib/lowlib was installed in the process but was then renamed to /lib/lowlib.old, and a new object module having the pathname /lib/lowlib was created. If the loader is then told to load a program requiring /lib/lowlib, it will detect that the file system object (currently) having that pathname is *not* installed in the process, even though earlier it did install a library that was (then) named /lib/lowlib.

In general, it is a good idea when specifying pathname to specify the *absolute* pathname of the object file. If you specify a relative pathname and then move the output object file to another directory, the loader will not be able to locate the object file. (See the "–BDIR" listing earlier in this encyclopedia for the distinction between absolute and relative pathnames.)

EXAMPLES

Consider an object file named main.bin that calls a function located in object file f.bin. Let's examine the ways in which the two object files can be associated.

First, you could bind them and run the resulting object file as follows:

```
$ bind main.bin f.bin  -binary my_program
$ my_program
```

A second method is to install f.bin and execute main.bin. The loader will resolve external references at runtime.

```
$ inlib f.bin
$ main.bin
```

A third method involves the -INLIB binder option. If we issue the following two commands, the loader will install f.bin when you invoke my_program:

```
$ bind main.bin -inlib f.bin  -binary my_program
$ my_program
```

Now let us examine the -NOINLIB option. Consider the following commands:

```
$ bind a.bin -inlib b.bin -binary lev1 (Use -inlib)
$ lev1                                 (Test lev1 with b.bin installed)
$ bind lev1 -noinlib b.bin  b.bin c.bin -binary lev2  (Use -noinlib)
$ lev2                        (Test lev2 with b.bin bound instead of installed.)
```

-LOCALSEARCH, -NOLOCALSEARCH -- Controls the order in which the binder will search through library files to satisfy unresolved external symbols.

FORMAT

-LOCALSEARCH
-NoLOCALSEARCH

DESCRIPTION

The -LOCALSEARCH and -NOLOCALSEARCH options control the way the binder searches through a library file. Before reading this description, you should see Section 3.3 which explains the default method that the binder uses to search through library files.

Consider what happens if you specify -NOLOCALSEARCH (the default). In this case, when the binder scans a library file in order to resolve external references, it makes a *single* pass through the library file. The binder loads those object modules that satisfy one or more unresolved references. The binder then moves on to the next library file you specified on the command line, if any. After the binder has scanned all the library files on the command line, it rescans them (in their original order) if any unresolved external references remain. The binder continues scanning the libraries until no new external references need to be resolved.

The search pattern described above sometimes causes the wrong object module to be loaded from a library file. For example, suppose a library file contains an object module that makes an external reference to a global symbol. However, the global symbol is defined by an object module that appears *earlier* in the same library. In this case, the binder may not load this object module on this pass through the library file. Ordinarily, this is not a problem because the binder will eventually rescan the library file. However, it is a problem when another object module *in a different library file* also happens to define the global symbol. In this case, the binder may load the wrong object module by accident. The purpose of -LOCALSEARCH is to prevent a faulty loading.

If you specify -LOCALSEARCH, instead of making a *single* pass over a library's object modules before moving on to the next library, the binder makes *multiple* passes over the library's object modules, moving on to the next library only after a pass results in no new object modules being loaded from the library. Therefore -LOCALSEARCH causes the binder to resolve as many external references as possible using object modules from the current library file before scanning the next library file. New unresolved references may arise from loading object modules from the library file.

Use -LOCALSEARCH to ensure that wherever possible, the binder resolves intra-library external references using global symbols defined by object modules in the *same* library. Without -LOCALSEARCH, you risk resolving external references with global symbols that happen to have the same name in object modules contained in *other* library files.

EXAMPLES

Consider the following information regarding some object files:

- Object file `main.bin` makes an external reference that can be resolved by an object module named `circle`.
- Object module `circle` is contained in library `mathlib1`. `Circle` makes an external reference that can be resolved by an object module named `subcircle`.
- Two different versions of object module `subcircle` exist -- one in `mathlib1` and the other in `mathlib2`. We want to load the version stored in `mathlib1`.

If we issue the following command:

```
$ bind main.bin mathlib1 mathlib2 -binary myprogram -nolocalsearch
```

then the binder will load the version of `subcircle` stored in `mathlib2`. Why? Consider the search path. The binder first loads `main.bin`. Then the binder makes a single pass through `mathlib1` and loads `circle` (as shown in Figure 2-1). (Since the desired version of `subcircle` precedes `circle` in the library, the binder will not load it.)

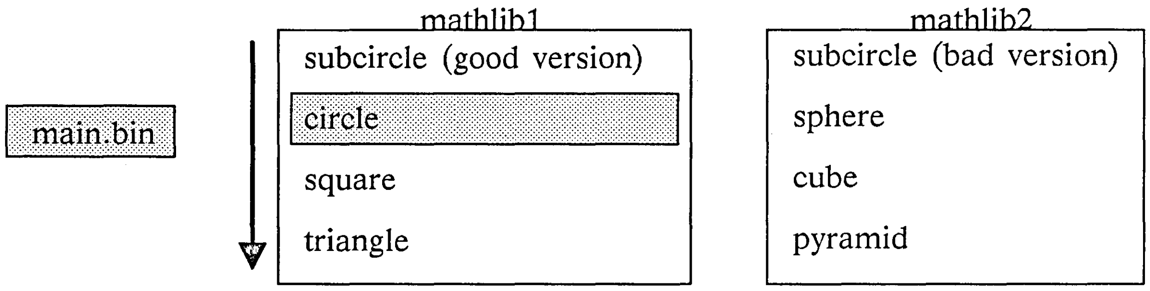

Figure 2-1. -NOLOCALSEARCH option. Beginning of search.

Next, the binder searches through `mathlib2` and therefore loads `subcircle` (as shown in Figure 2-2).

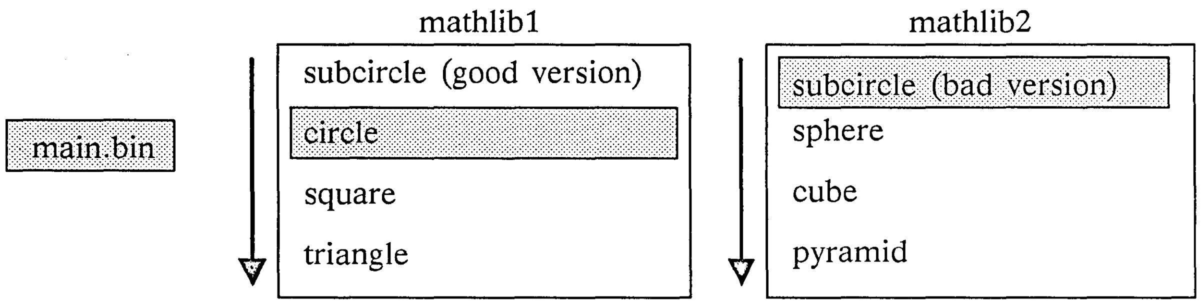

Figure 2-2. -NOLOCALSEARCH option. End of search.

Now consider what happens if we use the -LOCALSEARCH option as follows:

```
$ bind main.bin mathlib1 mathlib2 -binary myprogram -localsearch
```

In this case, the binder loads the version of `subcircle` stored in `mathlib1`. First the binder loads main.bin. Then it scans mathlib1 as shown in Figure 2-1. At the end of this first pass of mathlib1, the external reference to subcircle is still unresolved; therefore, the binder rescans mathlib1. During the second pass over mathlib1, the binder loads the good version of subcircle. The binder keeps rescanning `mathlib1` until no new external references can be resolved. (Therefore, the binder makes a total of three passes over mathlib1). Finally, the binder scans mathlib2. Figure 2-3 illustrates the search order.

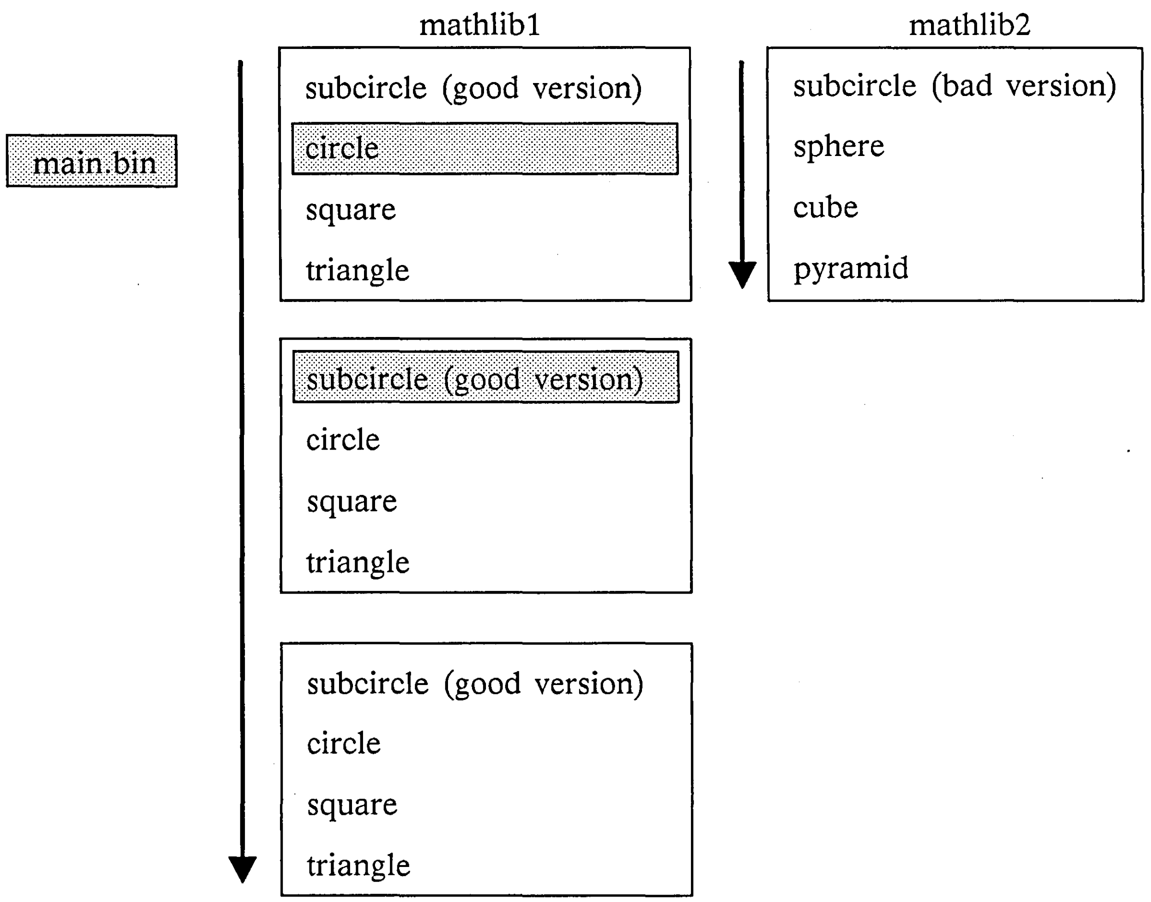

Figure 2-3. -LOCALSEARCH option. End of search.

-LOOKSECTION, -NOLOOKSECTION, -MARKSECTION, -UNMARKSECTION -- Controls the sharing of data sections between an executing object file and an installed library.

FORMAT

```
                      pick one from
                      this column
-LOOKSection       |                  |
-NOLOOKSection     |  section_name    |
-MARKSection       |  -ALL            |
-UNMARKSection     |                  |
```

ARGUMENTS

section_name	The name of one section defined by an input object module appearing earlier in the command line. That is, you must place the option *after* the section has been defined.
-ALL	The keyword -ALL. By specifying -ALL, the option applies to all sections in subsequent object modules appearing on the command line. -ALL affects all sections (with the correct attributes) in all subsequent object modules. Therefore, position the option *before* the object files that define the sections.

DESCRIPTION

These options only affect programmers who are creating their own installed libraries; if you are not developing object files to install, then you can ignore these options. You use these options to control the sharing of data at runtime between a section in a non-installed object file and a section in an installed library.

Use the -LOOKSECTION option to set the LOOKSECTION attribute; use the -MARKSECTION option to set the MARKSECTION attribute. The -NOLOOKSECTION and -NOMARKSECTION turn off the LOOKSECTION and MARKSECTION attributes, respectively. (See Appendix B for a description of all attributes.) If the following two conditions are both true, then the section shares data:

- The LOOKSECTION attribute of a section in a non-installed object file is set.
- The MARKSECTION attribute of a section in an installed library is set.

However, if *either* condition is not true, then the two sections do not share data. By default, both the LOOKSECTION attribute and the MARKSECTION attribute are off.

Data Sharing Between Two Installed Libraries

You can also use these options to permit (or to prevent) two installed libraries to share a data section. You do this by setting the MARKSECTION attribute on the section from the object file to be installed first and the LOOKSECTION attribute on the section from the object file to be installed subsequently. In addition, if you don't know which section is going to be installed first, you can cover all possibilities by setting both attributes on the same section in the object file.

Refer to the discussion of INLIB in Chapter 4 for more information about installed libraries.

Eligible Sections

The four options affect only those sections having all three of the following attributes:

- data
- overlay
- read/write

In other words, the binder ignores the option if the specified section does not have all the specified attributes. To create such a section in FORTRAN, you define a COMMON area. To create such a section in Pascal, you define a named section by putting a name in parentheses just after the reserved word VAR. To create such a section in the C language, just define a global variable. (For details, see the *DOMAIN Language Reference* manuals for FORTRAN, Pascal, and C.)

EXAMPLE

Suppose you created two object files (`a.bin` and `b.bin`) which each define a section named `c_array`. Further suppose that the `c_array` section has the data, overlay, and read/write attributes. Finally, you want `b.bin` to be part of an installed library, and `a.bin` to be non-installed.

If you want the installed library to share data in `c_array` with the executing program, then bind in the following manner:

```
$ bind  b.bin  -marksection c_array   -binary to_be_installed
$ bind  a.bin  -looksection c_array   -binary user_program
```

-MAKERS -- Displays the version numbers of the compilers, binders, etc. used to create the output object file.

FORMAT

-MAKers

DESCRIPTION

Use the -MAKERS option to learn the version numbers of the utilities that built the input object files.

EXAMPLE

```
$ bind ab c.bin -b abc -makers
This object was made by the following:
   cc, Rev 4.52, Date: 1986/09/04 15:05:11 EDT (Thu)
   bind, Rev 4.36, Date: 1986/07/30 15:47:14 EDT (Wed)
All Globals are resolved.
```

–MAP -- Causes the binder to print a load map.

FORMAT

–MAP

DESCRIPTION

Use the –MAP option to learn all sorts of information about the input and output object modules. The –MAP option produces a header, a section map, and a global map. If you only want the section map, specify the –SECTIONS option instead of –MAP. If you only want the global map, specify the –GLOBALS option instead of –MAP.

By default, the binder sends the listing to standard output. If you want to redirect the listing to a file, use the greater-than sign (>). For instance, the following example sends a map to file `ties.map`:

```
$ bind ties.bin  -MAP  -binary ties  >ties.map
```

EXAMPLE

```
$ bind a.bin b.bin c.bin -mark nick -binary abc -map (bind and create a map)
A P O L L O  Object Module Binder   5.03
1986/09/11 11:23:15 EDT (Thu)
File Name = abc
Module_Name = A_C  Version = 0.00
Start Address = 00000018 in Section    2

This object was made by the following:
   bind, Rev 5.03, Date: 1986/08/27 13:13:15 EDT (Wed)
   cc, Rev 4.52, Date: 1986/09/04 15:05:11 EDT (Thu)
Section Map:
 Id    Size     Name     Modules               Attributes
   1 000000FC procedure$                        R/O Concat Instr Long-aligned
              00000000 000000AC A_C
              000000AC 0000002C B_C
              000000D8 00000024 C_C
   2 00000120 data$                             Concat Data Zero Long-aligned
              00000000 000000CC A_C
              000000CC 00000028 B_C
              000000F4 0000002C C_C
   3 0000007E debug$                            R/O Concat Data Long-aligned
              00000000 0000004A A_C
              0000004A 00000018 B_C
              00000062 0000001C C_C
   4 0000000C big                               Ovly Mixed Data Look_installed Zero Long-aligned
   5 00000006 str                               Ovly Mixed Data Look_installed Zero Long-aligned
   6 00000011 nick                              Ovly Mixed Data Look_installed Zero Long-aligned
   7 00000190 rachel                            Ovly Mixed Data Look_installed Zero Long-aligned

Global Map:
  Offset  In Section  Name
 00000018       2    <apollo_c_startup>
 000000D0       2    b
 00000000       4    big
 000000F8       2    catch
 0000002C       2    main
 00000000       6    nick                       Marked
 00000000       7    rachel
 00000000       5    str
All Globals are resolved.
No Errors.
```

The Map Explained

The map can be divided into three distinct areas:

- Header
- Section Map
- Global Map

We examine these areas individually.

The Header

```
A P O L L O Object Module Binder    5.03
1986/09/11  11:23:15 EDT (Thu)
```

5.03 is the version number of the binder utility used in the example. Your binder version number may vary. The second line shows the year/month/day and hour:minute:second that the binding took place.

```
File Name = abc
Module_Name = A_C  Version = 0.00
Start Address = 00000018 in Section     2
```

abc is the name of the output object module. (See the "-MODULE" listing later in this section for information on object module names.)

The version number of abc is 0.00. (See the "-SET_VERSION" listing for information on version numbers.)

Start address refers to the address of the first instruction that is executed at runtime. The start address in our sample is the instruction located 18_{16} bytes past the beginning of section 2. You control the start address through your source code or through the -ENTRY binder option. If your source code is written in Pascal then the start address corresponds to the first executable instruction from the source file with the heading "PROGRAM." If your source code is written in C, then the start address corresponds to the first executable instruction from the source file in the "main()" function. If your source code is written in FORTRAN, then the first executable instruction in your main program will correspond to the start address. The start address in our example comes from the object module named A_C. (Refer to Section 3.4 of Chapter 3 for details on start address.)

```
This object was made by the following:
    bind, Rev 5.03, Date: 1986/08/27 13:13:15 EDT (Wed)
    cc, Rev 4.52, Date: 1986/09/04 15:05:11 EDT (Thu)
```

By default, the -MAP option lists the -MAKERS option information. The -MAKERS option tells you what compiler, binder version, etc. was used to create the object modules. Refer to the "-MAKERS" listing earlier in this encyclopedia for more information.

The Section Map

Since the section map can also be generated by the -SECTIONS option, we describe this map in the "-SECTIONS" listing later in this encyclopedia.

The Global Map

Since the global map can also be generated by the -GLOBALS option, we describe this map in the "-GLOBALS" listing earlier in this encyclopedia.

-MARK, -ALLMARK, -ALLKEEPMARK, -UNMARK, -ALLUNMARK -- Marks, unmarks, or preserves a mark on one or more global symbols.

FORMAT

```
       | You must specify one of |          | You must specify one of |
       | these two as arguments: |          | these two as arguments: |
       |                         |          |                         |
-MARK  |    global_symbol        | -UNMARK  |    global_symbol        |
       |    -ALL                 |          |    -ALL                 |
```

-ALLMARK (a synonym for -MARK -ALL)   **-ALLUNMARK** (a synonym for -UNMARK -ALL)

-ALLKEEPMARK

ARGUMENTS

global_symbol	You must specify the name of one global symbol. The global_symbol must have been *previously* defined by one and only one input object module on the binder command line.
-ALL	By specifying -ALL, you tell the binder to mark or unmark every global symbol in every input object module appearing *after* the -ALL option on the binder command line.

DESCRIPTION

By default, compilers **mark** all global symbols in the output binary file, and the binder **unmarks** all global symbols in the output object module it creates. In some situations, you may want to unmark a marked symbol or vice-versa, and we provide the -MARK, -UNMARK, -ALLMARK, -ALLUNMARK, and -ALLKEEPMARK options to do just that. The marking or unmarking of a symbol is important in the following two situations only:

- A binding operation in which two or more input object modules define the same global symbol. (Usually, this only happens when you perform multilevel binding. See Section 2.2.1 for an explanation of multilevel binding.)
- A binding operation in which the output object file will be installed as an installed library.

Let's now consider the first situation. When two or more input object files define the same global symbol, the following occurs:

- If the symbol is marked in only one file, then the binder uses that definition.
- If the symbol is marked in more than one file, then the binder uses the first marked symbol it encounters and then issues a "Multiply Defined Global" warning.
- If the symbol is unmarked in every file, then the binder uses the first definition encountered.

Now let's consider the second situation, namely, how marking affects installed libraries. An unmarked global symbol in an installed library cannot resolve an outstanding external reference, but a marked global symbol in an installed library can. Therefore,

- If you install an object file produced by the compiler (as opposed to the binder), then its global symbols can resolve outstanding external references at runtime.

- If you install an object file produced by the binder, then the global symbols cannot resolve outstanding external references at runtime unless you mark them when you bind.

The Five Options

Here is the distinction between the options:

- Use –MARK global_symbol to mark one global symbol. Place the option on the command line at some point after the global symbol has been defined by an input object file.
- Use –MARK –ALL or –ALLMARK to mark all global symbols defined by input object modules that appear after the option on the bind command line.
- Use –ALLKEEPMARK to preserve any existing marks on global symbols. The option only influences global symbols defined in input object files placed after the option on the bind command line.
- Use –UNMARK global_symbol to unmark one global symbol. Place the option on the command line at some point after the global symbol has been defined by an input object file.
- Use –UNMARK –ALL or –ALLUNMARK to unmark all global symbols defined by input object modules that appear after the option on the bind command line.

EXAMPLES

This section contains four examples demonstrating the various marking options. In all the examples, we rely on the following information:

- We created five object files (`a.bin`, `b.bin`, `c.bin`, `d.bin`, and `e.bin`) with a DOMAIN compiler.
- `a.bin` makes an unresolved external reference to symbol `earth`.
- `b.bin` and `c.bin` each define `earth` as a global symbol. Since the compiler created b.bin and `c.bin`, `earth` is a marked global symbol in both files.
- `d.bin` and `e.bin` neither define nor refer to `earth`.

Example 1

Consider the following bind command line:

```
$ bind a.bin b.bin c.bin  -binary abc
?(bind) Warning: "earth" Multiply Defined Global
All Globals are resolved.
```

The binder issued a warning because `earth` was marked in both `b.bin` and `c.bin`. Since it was marked twice, the binder resolves the unresolved external reference with the global symbol in `b.bin` since it appears first.

Example 2

Consider the following multilevel binding:

```
$ bind  b.bin d.bin      -binary lev1   (EARTH is unmarked in lev1)
$ bind  c.bin e.bin      -binary lev2   (EARTH is unmarked in lev2)
$ bind  a.bin lev1 lev2  -binary lev3
```

By default, the binder unmarks all global symbols when it creates the output object file. Therefore, the binder unmarks earth in lev1 and lev2. When creating lev3, the binder resolves the external reference from a.bin with the first occurrence of global symbol earth (from lev1 which was originally from b.bin).

Suppose you want to ensure that the binder resolves the external reference to earth with the global symbol earth stored in c.bin. To accomplish this, you must mark earth in c.bin (and unmark it in b.bin). So the sequence would look like this:

```
$ bind b.bin d.bin -unmark earth -binary lev1  (earth is unmarked in lev1)
$ bind c.bin e.bin   -mark earth -binary lev2  (earth is marked in lev2)
$ bind a.bin lev1 lev2          -binary lev3
```

The -unmark earth option in the first binder command is not necessary since earth will be unmarked by default.

Example 3

Object file b.bin was created by a compiler; therefore, earth is marked. If you issue the following INLIB command:

```
$ inlib b.bin
```

then earth will be accessible to running programs since it is marked in b.bin. However, if you try to install a bound object file, as in the following example:

```
$ bind   b.bin d.bin  -binary bound_file
$ inlib  bound_file
```

then earth will be inaccessible to running programs because it is unmarked in bound_file. If you want earth to be accessible to running programs, you should mark it as in the following example:

```
$ bind  b.bin d.bin  -mark earth -binary bound_file
$ inlib  bound_file
```

Example 4

The -ALLKEEPMARK preserves a mark that would otherwise disappear as the result of a multilevel binding. For example, in the following series of commands, earth is marked in lev1, but then becomes unmarked in lev2 (since the -MARK option was not specified in the second binder command):

```
$ bind b.bin -mark earth d.bin -binary lev1  {EARTH is marked in lev1}
$ bind lev1 e.bin -binary lev2               {EARTH is unmarked in lev2}
$ inlib lev2
$ a.bin      {External reference to EARTH cannot be resolved at runtime.}

... (runtime errors)
```

We correct the problem in the following series of commands simply by using an -ALLKEEPMARK option in the second binder command line:

```
$ bind b.bin -mark earth d.bin -binary lev1  {EARTH is marked in lev1}
$ bind -allkeepmark lev1 e.bin -binary lev2  {EARTH remains marked in
                                              lev2}
$ inlib lev2
$ a.bin     {External reference to EARTH can be resolved at runtime.}

... (no runtime errors)
```

-MERGEBSS -- Combines the sections generated by C global variables into one section named BSS$.

FORMAT

-MERGEbss

DESCRIPTION

Use this option to merge every section that corresponds to a C global variable into a single section named BSS$.

By default, the C compiler creates a new section for each global variable. The section name is the same as the global variable name. Use the -MERGEBSS option to merge all these sections into one section. You can greatly reduce the number of sections, and in many cases, improve load performance by using this option.

The C compiler gives the following attributes to the sections it creates for global variables: Ovly, Mixed, Data, Look_installed, Zero, and Long-aligned. (See Appendix C for a description of each attribute.) Normally, the -MERGEBSS option merges together all named sections having these attributes; however, this is not always the case. If a named section has these attributes, and an installed library available in the current shell contains a section by the same name that is visible to programs run in that process, then the binder does *not* merge the section into the BSS$ section. It instead assumes that the program intends to "share" global data with the installed library at excecution time.

When you use multilevel binding (see Section 2.2.1 for details) to develop a program, you should not use the -MERGEBSS option until the final bind.

By default, at SR9.5, the DOMAIN/IX 'ld' command and /bin/cc merge sections as described for -MERGEBSS. Use the 'r' flag (of either utility) to suppress this merging.

EXAMPLES

Consider the following sample C source code files:

Contents of file "hi.c"

```
int x = 5;
char rachel[] = {"Hello"};
extern void f()

main()
{
   printf("%d\n", x);
   printf("%s\n", rachel);
   f();
}
```

Contents of file "ho.c"

```
extern int x;


void f()
{
   printf("%d\n", x * 10);
}
```

Suppose we compile them. If we bind the resulting object files as follows, the binder will create a section named x to contain variable x and a section named rachel to contain variable rachel:

```
$ bind hi.bin ho.bin  -binary hideeho
```

However, if we bind the object files with the -MERGEBSS option as follows, the binder will create a section named BSS$ which will contain variables x and rachel:

```
$ bind hi.bin ho.bin  -mergebss -binary hideeho
```

-MESSAGES, -NOMESSAGES -- Directs the binder to report or suppress informational messages at the end of a successful binder session.

FORMAT

-MESsages (which can also be abbreviated to **-MSGS**)
-NOMESsages (which can also be abbreviated to **-NMSGS**)

DESCRIPTION

The binder prints two kinds of "informational" messages. The first informational message is

```
All Globals are resolved.
```

The second informational message is a report of the number of errors and warnings encountered during the binder session; for example:

```
2 Errors; 1 Warning
```

If there were no errors and no warnings, the binder does not print anything.

Use the -MESSAGES option to direct the binder to continue printing informational messages. Use the -NOMESSAGES option to suppress printing informational messages. -MESSAGES is the default.

EXAMPLES

```
$ bind a.bin b.bin c.bin -ruff -binary ab -nomessages
?(bind) Warning: "earth" Multiply Defined Global
        Input file "c.bin"
?(bind) Error: Unknown Command Ignored
        Input file "c.bin"
        Cmd = "-RUFF"
                                   (no informational messages)

$ bind a.bin b.bin c.bin -ruff -b ab -messages
?(bind) Warning: "earth" Multiply Defined Global
        Input file "c.bin"
?(bind) Error: Unknown Command Ignored
        Input file "c.bin"
        Cmd = "-RUFF"
All Globals are resolved.          (informational message)
1 Error; 1 Warning.                (informational message)
```

-MODULE -- Lets you specify a nondefault name for the output object module.

FORMAT

-MODule new_module_name

ARGUMENTS

new_module_name	Here's where you specify the new name for the output object module.

DESCRIPTION

By default, the binder uses the name of the first input object module it encounters as the name of the output object module. Use –MODULE to specify a nondefault name for the output object module.

–MODULE is particularly useful when you are preparing an object module to be passed on to the librarian. Since the librarian won't let you add a module if there is already a module with that name in the library, you can change the object module's name with –MODULE. Otherwise, the name of an object module has no effect on program execution. Don't confuse the object module's name with the name of the file that the object module is stored in. The –MODULE option has no effect on the filename.

You can find the name of the output object module by using the –MAP option.

EXAMPLES

Suppose the name of the object module stored inside file `math1.bin` is `real_math`. Therefore, if you issue the following command line, the binder names the output object module `real_math`:

```
$ bind math1.bin math2.bin -binary math
```

However, suppose you want the output object module to be called `double_real_math`. To accomplish this, you would issue the following command line:

```
$ bind math1.bin math2.bin -module double_real_math -binary math
```

-MULTIRES, -NMULTIRES, -NOMULTIRES -- Reports or suppresses errors if the binder finds multiple resolution of global symbols in library files.

FORMAT

-MULTIRES

-NMULTIRES
-NOMULTIRES $\Big>$ *these are synonyms.*

DESCRIPTION

The -MULTIRES option causes bind to report a particular error; the -NMULTIRES or -NOMULTIRES options causes the binder to suppress this error. The error in question is:

```
?(bind) Error: Multiple resolutions are possible for implicitly resolved
symbol
```

which means that more than one object module in a library file can resolve an unresolved external symbol.

Because -NOMULTIRES supppresses errors, by using this option you risk accidentally binding the wrong modules from a library file.

-NOMULTIRES is on by default.

EXAMPLES

Suppose that object file `a.bin` contains an unresolved external symbol named `B` which can be resolved by two different modules in library `mylib`. Compare the following two bind command lines:

```
$ bind a.bin mylib -multires -binary abcd
?(bind) Error: Multiple resolutions are possible for implicitly resolved
symbol
        Input file "mylib"
       Module name "c_c"
       Global name "B"
All Globals are resolved.
1 Error.

$ bind a.bin mylib -nomultires -binary abcd
All Globals are resolved.
```

-QUIT -- Causes an immediate exit from an interactive binder session.

FORMAT

-Quit

DESCRIPTION

The -QUIT option causes an immediate exit from the binder. The binder closes all input and output files but does not complete processing. The binder does not produce an output object module. However, if an existing object file has the pathname that the binder would have created, then the binder changes the name of the existing file by appending .bak to it (as described in Section 2.2.4).

-READONLYSECTION -- Changes the read/write attribute of a specified section to read-only.

FORMAT

-READONLYsection section_name

ARGUMENTS

section_name	The name of a section previously defined by an input object file. The section must have the read/write attribute in every input file that contains the section.

DESCRIPTION

Use the -READONLYSECTION option to change the read/write attribute of a specified section to read-only. A section with the read/write attribute is not write-protected, but a section with the read-only attribute is write-protected. Here is a list of read/write sections that you may want to change into read-only sections:

- In FORTRAN, any COMMON blocks or other COMMON sections whose contents are initialized by DATA statements and are not modified at runtime.
- In C, any global variable (assuming that the -MERGEBSS option was not used).
- In Pascal, any "named variable section". You create a named variable section by putting a section name in parentheses following VAR; for example:

```
VAR (a_named_section)
   X : INTEGER;
   Y : CHAR;
```

Here are three advantages that read-only sections have over read/write sections:

- Read-only sections are mapped to memory rather than copied by the loader. Therefore, the system performs less disk I/O.
- A read-only section cannot be overwritten or modified inadvertently.
- A read-only section does not require a backing store (i.e., some disk swapping space).

EXAMPLE

Suppose that an object module stored inside object file parser.bin contains a read/write section called parser_tables. To change parser_tables to read-only, you would issue the following command:

```
$ bind  main.bin parser.bin -readonlysection parser_tables -binary mlc
```

-SECTIONS -- Displays a section map.

FORMAT

-SECtions

DESCRIPTION

This option causes the binder to display a section map, which is a subset of the listing produced by -MAP. Use this option if you want information about sections but don't want the other information that comes with -MAP.

Note that the binder prints the section map based on the object modules preceding -SECTIONS on the command line. In other words, the section map that the binder produces depends on the position within the command line of -SECTIONS.

EXAMPLE

```
$ bind a.bin b.bin c.bin -binary abc -sections
Section Map:
 Id    Size     Name      Modules                  Attributes
   1 000000FC procedure$                           R/O Concat Instr Long-aligned
              00000000 000000AC A_C
              000000AC 0000002C B_C
              000000D8 00000024 C_C
   2 00000120 data$                                Concat Data Zero Long-aligned
              00000000 000000CC A_C
              000000CC 00000028 B_C
              000000F4 0000002C C_C
   3 0000007E debug$                               R/O Concat Data Long-aligned
              00000000 0000004A A_C
              0000004A 00000018 B_C
              00000062 0000001C C_C
   4 0000000C big                                  Ovly Mixed Data Look_installed Zero Long-aligned
   5 00000006 str                                  Ovly Mixed Data Look_installed Zero Long-aligned
   6 00000011 nick                                 Ovly Mixed Data Look_installed Zero Long-aligned
   7 00000190 rachel                               Ovly Mixed Data Look_installed Zero Long-aligned
All Globals are resolved.
```

The Section Map Explained

This section map tells us the following information:

- The **Id** number of each section -- this section map contains seven sections numbered 1 through 7.
- The total hexadecimal **Size** of each section -- for example, the total size of section 2 is 120_{16} bytes.
- The **Name** of each section -- for example, procedure$, data$, debug$, big, etc.
- The **Modules** comprising each section -- the binder supplies the following information under the Modules heading: the hexadecimal offset (within the section) of the contributing object module, the hexadecimal byte length (within the section) of the contributing object module, and the name of the contributing object module. For example, consider the following line of information:

  ```
  000000AC 0000002C B_C
  ```

  It shows that object module B_C contributed $2C_{16}$ bytes of data to section 2, and that these bytes are offset AC_{16} bytes from the start of section 2.

- The **Attributes** of each section -- a set of attributes characterizes each section. For example, section 2 has the concatenated, data, and long-aligned attributes. Appendix C explains what all these attributes mean.

-SET_VERSION -- Specifies the version number of the output object module.

FORMAT

-SET_VERsion nnnnn.mmmmm

ARGUMENTS

nnnnn.mmmmm	Is the version number of the output object module. You can specify any positive integer less than 65535 on either side of the decimal point. If you specify only one digit after the decimal point, the binder will precede this digit with a 0. For example, if you specify 5.7, the binder will give the output object file a version number of 5.07.

DESCRIPTION

Object files produced by a compiler do not carry a version number; however, object files produced by the binder do. A version number is simply two integers separated by a decimal point that you can use to help you distinguish between different versions of the same program. The default version number of an object file is 0.0. However, you can change this default number with the -SET_VERSION option.

The binder uses the following rules to determine the version number of the output object file:

- If you specify the -SET_VERSION option, the output object file will carry the version number specified by the option.
- If you do not specify the -SET_VERSION option, then the output object file will carry the version number of the first input object file that has a version number other than 0.0.
- If you do not specify the -SET_VERSION option and none of the input object files have a version number other than 0.0, then the binder sets the version number to 0.0.

Use the binder map (generated by the -MAP option) to find the actual version number generated by the binder.

EXAMPLES

```
$ bind a.bin b.bin c.bin -binary abc
```
(version number of abc = 0.0)

```
$ bind one.bin two.bin -set_version 10.20 -binary my_program
```
(version number of my_program = 10.20)

```
$ bind my_program three.bin -binary our_program
```
(version number of our_program = 10.20)

-SORTLOCATION, -SORTNAMES -- Sorts the list of global symbols in a global map.

FORMAT

-SORTLocation
-SORTNames

DESCRIPTION

These options affect the global symbols listing generated by the -MAP or -GLOBALS options. If you use -SORTLOCATION, the binder sorts the list of global symbols numerically, by section number and offset. If you use -SORTNAMES, the binder sorts the list of global symbols alphabetically, by name.

-SORTNAMES is the default.

EXAMPLES

```
$ bind a.bin b.bin c.bin -binary abc -sortlocation -globals
Global Map:
  Offset  In Section  Name
 00000018       2    <apollo_c_startup>
 0000002C       2    main
 000000D0       2    b
 000000F8       2    c
 00000000       4    big
 00000000       5    str
 00000000       6    nick
 00000000       7    rachel
All Globals are resolved.
```

```
$ bind a.bin b.bin c.bin -binary abc -sortnames -globals
Global Map:
  Offset  In Section  Name
 00000018       2    <apollo_c_startup>
 000000D0       2    b
 00000000       4    big
 000000F8       2    c
 0000002C       2    main
 00000000       6    nick
 00000000       7    rachel
 00000000       5    str
All Globals are resolved.
```

-SYSTEM -- Lists as "Undefined Globals" those symbols that can be resolved by installed libraries.

FORMAT

-SYStem
-NOSYStem

DESCRIPTION

If you use the -SYSTEM option, the binder classifies as "Undefined Globals" any external reference that cannot be resolved by an input object module. In other words, the -SYSTEM option causes the binder to ignore the global symbols defined in installed libraries when reporting "Undefined Globals." You can use this option to verify that the binder is, in fact, referring to the expected specific global symbols defined in installed libraries. The -SYSTEM option is purely informational and has no effect on the output object module.

EXAMPLES

Compare the following two bind sessions. The first command line does not contain -SYSTEM, but the second one does.

```
$ bind a.bin b.bin -binary myprog
Undefined Globals:
   catch                              First referenced in A.BIN

$ bind -system a.bin b.bin -binary myprog
Undefined Globals:
   catch                              First referenced in A.BIN
   printf                             First referenced in A.BIN
   scanf                              First referenced in A.BIN
   unix_$main                         First referenced in A.BIN
```

In the first session, an "Undefined Global" was any external reference that could not be resolved by another input object file or an object module in an installed library. In the second session, an "Undefined Global" was any external reference that could not be resolved by another input object file. Notice that the output object file, myprog, is the same for both sessions.

-SYSTYPE -- Override the systype for which the input object modules were compiled, and stamp the output object module with the given systype.

FORMAT

	You must specify one of these five as an argument:
	bsd4.1
	bsd4.2
-SYSTYPE	sys3
	sys5
	any

ARGUMENTS

bsd4.1	To specify the Berkeley 4.1bsd environment.
bsd4.2	To specify the Berkeley 4.2bsd environment.
sys3	To specify the AT&T System III environment. (This is the default systype.)
sys5	To specify the AT&T System V release 2 environment.
any	To specify that the program is independent of a particular environment.

DESCRIPTION

The -SYSTYPE option is of importance primarily to C programmers or to programmers intending to run the output object module under DOMAIN/IX. An object module's systype, which is normally set by the C compiler, affects its runtime behavior. The systype determines which set of functions are called and makes sure that the proper calling conventions are used. This is important since different DOMAIN/IX environments may have functions with the same name but with different semantics or calling conventions.

By default, the binder propagates the systype of input object modules by stamping the output object module with the same systype. The binder reports an error if input object modules are stamped with conflicting systypes.

You can use the binder's -SYSTYPE option to specify a non-default systype for the output object module. You *must* use the -SYSTYPE option if input object modules are stamped with conflicting systypes, in order to suppress the error the binder would otherwise report.

NOTE: Be especially careful about using the systype "any". Most programs are *not* independent of a particular operating system version. If your target is the DOMAIN operating system, then the systype must be "sys3."

For more information about systypes and their effect on the C compiler and the runtime environment, refer to the *DOMAIN C Language Reference* or the *DOMAIN C Library (CLIB) Reference.*

-UNDEFINED, -NOUNDEFINED, -NUNDEFINED -- Displays or suppresses a listing of unresolved external references.

FORMAT

-UNDefined

-NOUNDefined
-NUNDefined > *These are synonyms.*

DESCRIPTION

Use the -UNDEFINED option to display a list of any unresolved external references in an interactive binder session. This is a very useful option because it can help you determine which if any object files you omitted from the binder command line.

The -NOUNDEFINED (or -NUNDEFINED) option suppresses the listing of undefined globals that the binder lists by default at the end of a binder command.

EXAMPLES

```
$ bind test1.bin
* test2.bin
* -undefined
Undefined Globals:

   simple_exp                First referenced in //OXY/B/TEST1.BIN
* test6.bin
* -undefined
All Globals are resolved.
* -end
```

In the preceding binder session, the first -UNDEFINED showed us that `test1.bin` made an unresolved external reference to `simple_exp`. Therefore, we added object file `test6.bin` to the binder command line (knowing that it resolves `simple_exp`). Before ending the session, we used -UNDEFINED a second time which confirmed that "`All Globals are resolved.`"

In the following example, notice how the -NOUNDEFINED option suppresses the listing of unresolved external references in the binder's final report:

```
$ bind a.bin d.bin -noundefined

$ bind a.bin d.bin
Undefined Globals:

   earth                            First referenced in A.BIN
```

-XREF -- Provides a cross-reference listing.

FORMAT

-XREF

DESCRIPTION

The -XREF option tells you which object modules and sections refer to other modules and sections. It also shows you which modules and sections define global symbols, and where those global symbols are resolved. This option allows you to see how object modules are using globally visible names.

-XREF can only provide cross-reference information on the files that come after it on the command line. Therefore, if you put -XREF at the end of the binder command line, the cross-reference will show nothing. Conversely, if you put it right after the command name BIND, -XREF will provide a cross-reference of every input object module.

See the next page for an example.

EXAMPLE

```
$ bind -xref a.bin mylib -binary my_program
All Globals are resolved.
Module Cross Reference
a_c    Compiled: 1986/09/10 15:08:38 EDT (Wed)
  Defined Globals:
    <APOLLO_C_STARTUP>  BIG                MAIN
  References To Globals:
    B  C
  References To Modules:
    b_c  c_c
  Sections:
    BIG          DATA$        DEBUG$       PROCEDURE$
b_c    Compiled: 1986/09/10 13:38:03 EDT (Wed)
  Defined Globals:
    B
  Referenced By Modules:
    a_c
  Sections:
    DATA$        DEBUG$       PROCEDURE$
c_c    Compiled: 1986/09/10 13:38:06 EDT (Wed)
  Defined Globals:
    C
  Referenced By Modules:
    a_c
  Sections:
    DATA$        DEBUG$       PROCEDURE$
Global Cross Reference
<APOLLO_C_STARTUP> Defined In :a_c
B Defined In :b_c
  Referenced By Modules:
    a_c
BIG Defined In :a_c
C Defined In :c_c
  Referenced By Modules:
    a_c
MAIN Defined In :a_c
Section Cross Reference
BIG
  Defined In Modules:
    a_c
DATA$
  Defined In Modules:
    a_c  b_c  c_c
DEBUG$
  Defined In Modules:
    a_c  b_c  c_c
PROCEDURE$
  Defined In Modules:
    a_c  b_c  c_c
```

Chapter 3
How to Use the Librarian

Use the librarian to create, edit, or describe a library file. A library file consists of one or more object modules collected together for easy access by the binder. Typically, you store object modules in a library file so that the binder will load a subset of them. This chapter details the following topics:

- How to invoke the librarian.
- How to create a library file.
- How the librarian analyzes command lines.
- How to spread a librarian command over multiple lines.
- How to imbed comments in a librarian command.
- How the binder analyzes library files.
- How the binder sometimes uses library files to determine program start address.
- How to use all the librarian options.

To learn about the role library files play in program development, see Chapter 1.

> NOTE: This chapter is about library files; it is not about DOMAIN Software Engineering Environment (DSEE) libraries or installed libraries.

3.1 Invoking the Librarian

Use the following format to invoke the librarian:

```
$ LBR   -CReate   library_pathname   object_pathname(s)   option(s)
        -UPDate
```

After the keyword LBR, you must enter either the keyword -CREATE or the keyword -UPDATE. Enter -CREATE if you are creating a library. Otherwise, enter -UPDATE (even if you just want a listing). The abbreviation for -CREATE is -CR and the abbreviation for -UPDATE is -UPD.

Following -CREATE or -UPDATE, you must enter a library_pathname. If you specified -CREATE, enter the pathname you want to create; this pathname must not already exist. If you specified -UPDATE, library_pathname must be the name of the library you want to work on. For -UPDATE, you must pick the pathname of an existing, valid library.

If you specified -CREATE, then following the library_pathname, you must specify at least one object pathname. Such an object can be either an object file (i.e., a file produced by either the compiler or the binder) or another library file. The object modules within these object_pathnames will form the contents of the created library file.

If you used -UPDATE, then you can optionally enter one or more object pathnames which the librarian will add to the existing library.

Finally, you can enter zero or more of the following options:

-DELETE	Deletes one object module from the library.
-EXTRACT	Extracts one object module from the library and optionally writes it to another file.
-LIST	Generates a library file map.
-MSGS, -NOMSGS	Tells the librarian to write or suppress purely informational messages.
-QUIT	Causes the librarian to ignore everything that appears after the option on the command line.
-REPLACE	Replaces an object module already stored in a library file or adds a new object module to the library file.
-SYSTEM, -NOSYSTEM	Affects the map generated by -LIST; does not affect the output library file in any way. These options control the manner in which the librarian reports unresolved external references.

NOTE: You can only use wildcards with the -REPLACE option or while adding new object files to a library. If you try to use wildcards in any other place in the command, the librarian issues an error message.

3.1.1 Creating a Library File: Examples

The following command line creates a library file named `mylib` containing the object modules stored in files `b.bin`, `c.bin`, and `d.bin`.

```
$ lbr -create mylib  b.bin c.bin d.bin
```

The following command builds a library named `mylib2`, from another library (`mylib`) and from all the object modules stored in `e.bin` and `a.bin`:

```
$ lbr -create mylib2 e.bin a.bin mylib
```

3.1.2 Order of Execution

The librarian executes options and prints warning messages as it encounters them. For example, consider the following command line:

```
$ lbr -update math.lib t.bin sine.bin -list g1.bin -replace cost.bin -list
```

The librarian executes the commands as they appear from left to right. That is, the librarian executes the commands in the following order:

1. The beginning of the command (`$ lbr -update math.lib`) tells the librarian that you intend to work on existing library file `math.lib`.
2. The librarian adds the object modules stored in `t.bin` and `sine.bin` to `math.lib`.
3. The -list option tells the librarian to list the contents of `math.lib` at that point.
4. The librarian adds the object module stored in g1.bin to math.lib.
5. The -replace option tells the librarian to replace an object module from math.lib with the object module stored in cos.bin.
6. The final -list option tells the librarian to list the contents of math.lib.

If your command line contains an error, the librarian stops executing at the error. Therefore, the portion of the command prior to the error is still executed. For example, suppose that `g1.bin` (in the previous example) had not contained a valid object module. In this case, the librarian adds `t.bin` and `sine.bin` to `math.lib`, lists (-LIST) the contents of `math.lib`, prints an error message, and returns to the Shell.

3.1.3 Spreading a Librarian Command Over Several Lines

If you want to spread a librarian command over more than one line, you must either:

- Put a hyphen (-) at the end of the first line.
- Enter the command LBR (and nothing else) as the first line.

To signal the end of a continued the librarian command, you must either put -END at the end of the command or leave the final line blank. For example, the following three librarian commands are equivalent. All three create a new library (`math.lb`) out of ten object modules:

```
$ lbr -create math.lb -
* add.bin  sub.bin  mult.bin  div.bin  exp.bin  e.bin
* log10.bin  ln.bin  sine.bin  cosine.bin -end
$
```

or

```
$ lbr -create math.lb -
* add.bin  sub.bin  mult.bin  div.bin  exp.bin  e.bin
* log10.bin  ln.bin  sine.bin  cosine.bin
*
$
```

or

```
$ lbr
* -create math.lb
* add.bin  sub.bin  mult.bin  div.bin  exp.bin  e.bin
* log10.bin  ln.bin  sine.bin  cosine.bin
*
$
```

3.1.4 In-Line Comments

You can put comments on your LBR command line. Simply enclose your comments in braces, as in the following example:

```
$ lbr
* -upd my.lib
* vec?*.bin {gather vector modules}
* plot.bin {vector plotting}
* {vector mapping modules:}
* mapa.bin
* map13.bin
* map.lib
* -list {generate a listing and finish} -end
```

3.2 Errors and Warnings

If the librarian detects a problem with the command line, it issues either an error message or a warning message. An error message indicates that the librarian could not perform the requested operation or that some error condition arose while the librarian was trying to perform the operation. In either case, the result is probably an unusable library file.

A warning message indicates that one of the following is true:

- The librarian could perform the requested operation, but the contents of the library file may not be what you were expecting.
- The librarian could not perform the operation, but the library file was not corrupted. Therefore, you can issue a corrected command on the original library file.

Appendix B contains a complete list of all librarian error and warning messages, and an explanation of the likely cause of the problem.

3.3 How the Binder Scans Library Files

Unlike many binder or linker products on other operating systems, the order of object modules in a DOMAIN library is relatively unimportant. You may be familiar with one or two pass binder or linker products in which it is impossible to make forward external references. However, using the DOMAIN binder, you can make both forward and backward external references to object modules in a library. That's because the DOMAIN binder makes as many passes as is necessary to resolve outstanding external symbols. The result is that, with one exception, the DOMAIN binder will create the same program file regardless of the arrangement of object modules inside the library. The one exception is that when more than one object module could satisfy an unresolved external symbol, the binder loads the first it encounters.

Here's how the binder scans to resolve external symbols. The binder starts scanning at the first object file or library file on the command line and moves to the right. As the binder scans, it automatically loads all object modules stored in object files and all -INCLUDEd object modules. Any of these object modules may contain external references and definitions. The binder first tries to resolve these external references and external definitions in object modules already loaded. When the binder reaches the end of the object files and libraries, it checks to see if there are any remaining unresolved external references. If there are none, the scan ends. However, if there are some unresolved external references, the binder rescans. On a rescan, the binder only searches library files. The binder searches library files in the order you entered them on the command line. On a rescan, the binder attempts to satisfy outstanding unresolved references by using unloaded modules from library files. This process continues until the binder determines either that all external references are resolved or that no further resolutions can be made.

The binder always scans libraries in the order presented on the command line. Within a library, the binder scans libraries in the order that they appear in the report generated by the -LIST option of the librarian.

3.4 Program Start Address

A start address is the first executable instruction of the output object file. Although the binder, not the librarian, determines the start address, we describe the process here because library files play an important role in the determination.

The binder calculates the start address from the possible start addresses defined by the input object files and library files. By default, you define a possible start address through the following source code:

- In FORTRAN, the possible start address is the first executable instruction in the main program unit.
- In Pascal, the possible start address is the first executable instruction in the source file that has the header "PROGRAM".
- In C, the possible start address is the first executable instruction in the "main()" function.

(If you don't want a default start addres, you can use the –ENTRY binder option to define a nondefault one.)

The binder uses the following rules to determine the start address of the output object module:

- If exactly one input object module defines a possible start address, then this becomes the start address of the output object module.
- If more than one input object module defines a possible start address, then the binder sets the start address to the first possible start address it encounters.
- If no input object module defines a possible start address, then the binder looks for a possible start address in the unloaded library object modules. The binder makes this search only if it would have to scan the libraries anyway to satisfy unresolved external references. That is, on the binder's first pass, it tries to find a possible start address in the input object files and –INCLUDEd library object modules. However, if none of them defines a possible start address, then the binder (concurrent with its search to satisfy unresolved external references) searches the unloaded library object modules for a possible start address. The binder will load the first library object module that defines a possible start address (even if it does not satisfy an outstanding external reference). If the binder resolves all external references prior to finding a possible start address, then it halts the search and leaves the output object module without a start address.

3.5 Detailed Descriptions of Each Librarian Option

The remainder of this chapter is devoted to detailed descriptions of each librarian option.

-DELETE -- Deletes one object module from the library file.

FORMAT

-DELete object_module_name

ARGUMENTS

object_module_name Specify the name of one object module stored in the library file.

DESCRIPTION

Deletes one object module from the library file. If you accidentally specify an object module that is not in the library file, the librarian issues a warning. Note that the librarian is case-sensitive to object_module_name.

EXAMPLE

The following command line removes an object module named `circle` from the library file `mylib`:

```
$ lbr -update mylib -delete circle
```

-EXTRACT -- Finds the named object module inside a library file and copies it to another file.

FORMAT

-EXtract object_module_name *-O pathname*

ARGUMENTS

object_module_name	Specify the name of one object module stored in the library file. This is the object module that you want to copy from the library. Note that the librarian is case-sensitive to object_module_name.
-O pathname	The -O pathname is optional. If you specify it, the librarian copies the object module into pathname. If you do not specify -O pathname, the librarian copies the object module to a file having the same name as the object module.

DESCRIPTION

Use the -EXTRACT option to make a copy of an object module stored inside a library file. You can write the object module to the pathname of your choice. The -EXTRACT option does not change the library file in any way.

EXAMPLES

The following command finds the object module named `circle` from the library and copies it to a file named `circle`.

```
$ lbr -update mylib -extract circle
```

The following command finds the object module named `circle` from the library and copies it to a file named peg.

```
$ lbr -update mylib -extract circle -o peg
```

-LIST -- Generates a library file map.

FORMAT

-List

DESCRIPTION

Writes a report of the library file contents to standard output. This report contains the name of each object module in the library file, with a list of section information and global declarations and references for each object module.

EXAMPLE

```
  lbr -update mylib -list              (Create a list)
A P O L L O  Object Module Librarian   2.09
1986/09/29 16:29:24 EDT (Mon)

Library file name = mylib
module  section  symbol                  bytes     offset   attributes
-----------------------------------------------------------------------
CIRCLE_C  timestamp: 1986/09/29 16:26:53 EDT (Mon)
        entered into library on: 1986/09/29 16:27:20 EDT (Mon)
        procedure$                      00000048  00000020  R/O Concat Instr Long-aligned
        data$                           00000024  00000000  Concat Data Zero Long-aligned
                 circle                           00000010
        debug$                          0000001C  00000068  R/O Concat Data Long-aligned
        rachel                          00000009  00000000  Ovly Data Look_installed Zero
Long-aligned
                 rachel                           00000000
-----------------------------------------------------------------------
SQUARE_C  timestamp: 1986/09/29 16:29:08 EDT (Mon)
        entered into library on: 1986/09/29 16:29:19 EDT (Mon)
        procedure$                      00000048  00000020  R/O Concat Instr Long-aligned
        data$                           00000024  00000000  Concat Data Zero Long-aligned
                 square                           0000000C
        debug$                          0000001E  00000068  R/O Concat Data Long-aligned
external references:
                 line
-----------------------------------------------------------------------
```

The Map Explained

This section explains the elements of the sample map.

```
A P O L L O  Object Module Librarian   2.09
1986/09/29 16:29:24 EDT (Mon)

Library file name = mylib
```

2.09 is the version number of the librarian utility. The second line shows the year/month/day and hour:minute:second that we executed the librarian. The next line shows the name of the file that the librarian is providing information about (`mylib`).

```
module   section  symbol                    bytes     offset    attributes
--------------------------------------------------------------------------

CIRCLE_C  timestamp: 1986/09/29 16:26:53 EDT (Mon)
         entered into library on: 1986/09/29 16:27:20 EDT (Mon)
         procedure$                        00000048  00000020  R/O Concat Instr
         data$                             00000024  00000000  Concat Data Zero
                  circle                             00000010


         debug$                            0000001C  00000068  R/O Concat Data
         rachel                            00000009  00000000  Ovly Data Look_
                  rachel                             00000000
```

Next, the map describes each of the object modules in the library file. A line of dashes (–) separates each input object module. For each input object module, the listing shows the object module's name, timestamp, and time entered into the library. For example, the first input object module is named `CIRCLE_C`. It was created (by a compiler or by the binder) on September 29 at 4:26 in the afternoon. `CIRCLE_C` became part of `mylib` on September 29 at 4:27 in the afternoon. Note that the name of the object module does not necessarily correspond to the name of the file originally containing it.

Next, the listing describes the following information:

- Under the heading "section" -- the map lists all the sections comprising the input object module. For example, `CIRCLE_C` consists of the `procedure$`, `data$`, `debug$`, and `rachel` sections.
- Under the heading "symbol" -- the map lists all the global symbols defined in each section. For example, `circle` is defined in the `data$` section.
- Under the heading "bytes" -- the map shows the hexadecimal size of each section in bytes. For example, the `procedure$` section is 48_{16} (= 72_{10}) bytes long.
- Under the heading "offset" -- the map shows the starting position of each section or global variable. For a section, the offset is the hexadecimal number of bytes that the section (or global variable) is offset from the beginning of the object module. For a global variable, the offset is the hex number of bytes that the global variable is offset from the beginning of the section that contains it. For example, the `data$` section starts at the very beginning of the object module, but the `procedure$` section is offset 24 bytes from the start of the object module. Furthermore, global variable `circle` is defined 10 bytes from the beginning of the `data$` section.
- Under the heading "attributes" -- the map shows the attributes that characterize the section. See Appendix C for a description of what these attributes mean.

```
external references:
                 line
```

- Under the heading "external references" -- the map lists all symbols referred to in the object module but not defined by the object module. In other words, the listing names all unresolved external references. For example, the map shows that `square` contains an external reference to `line`. You must resolve these external references at bind time or runtime in order for the program to execute properly. This map did not show any external references that could be resolved by global symbols in an installed library. However, we could have gotten a list of those references by using the –SYSTEM option prior to the –LIST option.

-MESSAGES, -NOMESSAGES -- Reports or suppresses a report on the number of errors and warnings encountered in a binder session.

FORMAT

-MESsages	(-MESSAGES can be abbreviated as -MES or -MSG)
-NOMESsages	(-NOMESSAGES can be abbreviated as -NOMES or -NMSGS)

DESCRIPTION

Use these two options to force the librarian to either report or suppress a summary of the number of errors and warnings that occurred in a librarian session. -MESSAGES (the default) forces the report, and -NOMESSAGES suppresses the report.

EXAMPLES

Compare the following two lbr command lines. In the first command line we used the -MESSAGES option to generate an informational summary.

```
$ lbr -messages -create mylbr foobar.bin
?(LBR) Error: CREATE option specified but named file already exists,
can't create.
       File name "mylbr"
?(LBR) Error: No library specified, no add done.

  2 Errors.   <--------------        (the informational summary)
```

But in this command line we used the -NOMESSAGES option to suppress an informational summary:

```
$ lbr -nomessages -create mylbr foobar.bin
?(LBR) Error: CREATE option specified but named file already exists,
can't create.
       File name "mylbr"
?(LBR) Error: No library specified, no add done.

              <--------------        (no informational summary)
```

-QUIT -- Causes the librarian to ignore everything that appears after this option on the command line.

FORMAT

-Quit

DESCRIPTION

The -QUIT option causes the librarian to ignore everything that appears after it on the command line. The librarian still attempts to process every part of the command that precedes -QUIT. This option is useful if you detect a mistake somewhere in the middle of a command, but you still want the librarian to execute the beginning. If the LBR command spans more than one line, then the line on which -QUIT appears will be the last.

EXAMPLE

```
$ dlf whylib
$ lbr -create whylib -
* square.bin circle.bin
* -quit triangle.bin
$
```

-REPLACE -- Replaces one or more object modules already stored in a library file or adds new object modules to the library file.

FORMAT

-REPLace pathname

ARGUMENTS

pathname	The pathname of an object file or library file.

DESCRIPTION

Use -REPLACE to replace one or more object modules stored in a library file, or to add new object modules to the library file. The librarian handles the -REPLACE pathname option in the following way.

1. The librarian reads the file stored in pathname to learn which object module(s) is stored there.
2. The librarian scans the library file to see if this object module(s) is stored in the library file.
3. If the object module is stored in the library file, the librarian deletes it from the library file and stores the object module from pathname in its place. If the object module is not stored in the library file, the librarian issues a warning and then adds the object module to the library file.

EXAMPLES

Suppose that an object module named `triangle` was stored inside `mylib`; however, you discovered a flaw in it. The source code for `triangle` is stored in file `d.pas`. Therefore, you correct the problems in `d.pas` and recompile it to create `d.bin`. Finally, to replace the defective `triangle` in `mylib` with the good triangle in `d.bin`, you issue the following command:

```
$ lbr -update mylib -replace d.bin
```

Now suppose that object file `e.bin` contains an object module named `rhombus`, and that no version of `rhombus` is stored in `mylib`. If we issue the following command, the librarian issues a warning but adds the object module to the library anyway.

```
$ lbr -update mylib -replace e.bin
?(LBR) Warning: Replace of module which is not in library, module is
added.
       File name "e.bin"
     Module name "RHOMBUS"
No Errors; 1 Warning.
```

Instead of the preceding command, we could have issued the following command which adds the contents of `e.bin` to the library without issuing a warning.

```
$ lbr -update mylib e.bin
```

-SYSTEM, -NOSYSTEM -- Affects the listing generated by -LIST.

FORMAT

-SYStem
-NoSYStem

DESCRIPTION

These two options affect the listing generated by the -LIST option. If you specify -SYSTEM, the librarian reports unresolved external references even if they can be resolved by an installed library. If you specify -NOSYSTEM, the librarian does not report unresolved external references if they can be resolved by an installed library.

These are purely informational options; neither one affects the output library file in any way.

-NOSYSTEM is the default.

EXAMPLES

Here we compare the affect of the -SYSTEM option to the -NOSYSTEM option. First, we study -SYSTEM. Notice how the following listing reports as "external references" a symbol that can be resolved by an installed library.

```
$ lbr -create whylib circle.bin square.bin -system -list
.
.          (We omitted irrelevant parts of the listing.)
.
-----------------------------------------------------------------------
SQUARE_C  timestamp: 1986/09/29 15:54:00 EDT (Mon)
          entered into library on: 1986/09/29 15:54:35 EDT (Mon)
          procedure$                    0000002C  00000020  R/O Concat Instr Long-aligned
          data$                         00000020  00000000  Concat Data Zero Long-aligned
                     square                        00000008
          debug$                        0000001C  0000004C  R/O Concat Data Long-aligned
external references:
                m$mis$111                          printf
-----------------------------------------------------------------------
```

Next, we examine the -NOSYSTEM option. Notice how it does not report the external references that -SYSTEM reported.

```
$ lbr -create whylib circle.bin square.bin -nosystem -list
.
.          (We omitted irrelevant parts of the listing.)
.
-----------------------------------------------------------------------
SQUARE_C  timestamp: 1986/09/29 15:54:00 EDT (Mon)
          entered into library on: 1986/09/29 15:54:35 EDT (Mon)
          procedure$                    0000002C  00000020  R/O Concat Instr Long-aligned
          data$                         00000020  00000000  Concat Data Zero Long-aligned
                     square                        00000008
          debug$                        0000001C  0000004C  R/O Concat Data Long-aligned
-----------------------------------------------------------------------
```

Chapter 4

Installed Libraries

An installed library is a set of one or more object modules that can only be accessed at runtime. (See Chapter 1 for an introduction to installed libraries.) There are five types of installed libraries:

- User-defined installed libraries
- System-defined installed libraries
- System-defined global libraries
- User-defined global library
- Object files installed with the -INLIB binder option

This chapter describes all five types.

NOTE: Most programmers can skip over this chapter. If, however, you want to create your own installed libraries, you will find this chapter quite useful.

4.1 User-Defined Installed Libraries

A user-defined installed library is a very useful alternative to loading object modules with the binder. Basically, the binder resolves external symbols at bind time (of course), but if you've created an user-defined installed library, the loader resolves external symbols at runtime.

A user-defined installed library is temporary; it lasts only as long as the shell process that created it. You create a user-defined installed library with the shell command INLIB. Invoking INLIB is simple: all you have to do is issue a shell command of the following format:

```
$ INLIB pathname
```

Pathname must be the name of an object file created by a compiler or the binder. *Don't be fooled by the term "installed library"; pathname cannot be the name of a library file created by the librarian.* See Appendix D for an example demonstrating the INLIB utility.

A user-defined installed library exists only within the process that invoked INLIB. That is, your program cannot successfully access an installed library from another shell process. When you terminate the shell process (with CTRL/Z), the installed library becomes uninstalled. You can use INLIB more than once in the same shell process to install more object files as installed libraries.

If the object file contains a "main program" (i.e., if it defines a start address), then INLIB executes the object file at installation time. This allows the installed object file to initialize static data. If the object file does not contain a "main program," then INLIB does not execute the object file at installation time.

> NOTE: If you used a compiler to create the object file you are installing, then all global symbols in the user-defined installed library are automatically accessible to running programs. However, if you used the binder to create the object file you are installing, then by default the global symbols in the installed library are *inaccessible* to running programs. To make these global symbols accessible to running programs, you must use the -MARK, -ALLMARK, or -ALLKEEPMARK option when you bind. See Chapter 2 for full details on these important binder options.

4.2 System-Defined Installed Libraries

System-defined installed libraries function identically to user-defined installed libraries; the only difference is that we write them, not you. You install them with the INLIB utility. After you install a system-defined installed library, you can only access it in the shell process in which you installed it.

Two examples of system-defined installed libraries are `/lib/gmrlib` and `/lib/d3mlib`. For example, to install /lib/gmrlib, you would issue the following command:

```
$ inlib /lib/gmrlib
```

4.3 System-Defined Global Libraries

When you boot a DOMAIN workstation, the DOMAIN system automatically generates installed libraries, called global libraries. As long as the software is installed, any program running on your workstation can access them. The installation of these global libraries is automatic and beyond user control.

An example of a system-defined global library is `/lib/ftnlib`.

4.4 The User-Defined Global Library

A user-defined global library is a hybrid of the other types of installed libraries. Like a system-defined global library, it is available to any program running in any shell at any time. But like a user-defined installed library, you, the user, write it and control it.

To create a user-defined global library, simply copy an object file to pathname `/lib/userlib.private`. Then exit from the Display Manager (DM) with the DM command EX. When you restart your workstation, the operating system generates a global library (i.e., an installed library automatically accessible to every process). If you find that you don't like this installed library, then you must remove it, shut down the DM, and restart. You can remove it by issuing the following command:

```
$ dlf /lib/userlib.private -du -f
```

When you create a process, the loader automatically executes the main program of the object file stored in `/lib/userlib.private` if there is one. This can slow down the creation of processes, so we recommend that your object file *not* contain a main program unless absolutely necessary.

4.4.1 Initializing Static Data in The User-Defined Global Library

The loader automatically initializes static data in the user-defined global library. However, the method of initialization depends on what section the static data is stored in.

If the static data is stored in the `data$` section, then the system initializes it to the specified values when you boot the workstation or restart the DM. The loader initializes the data the same way it would initialize static data for a running program. However, after initialization, the `data$` section (which normally has the read/write attribute) becomes a read-only section to prevent data corruption.

If the static data is stored in a section other than `data$`, then the loader ignores the values specified in source code. Instead, when you boot your workstation or create a process, the loader automatically sets the value of all static data in the section to zero. Each process has its own private copy of this section, and the virtual addresses the section occupies are the same for every process. Most importantly, the loader gives this section the read/write attribute. Therefore, although you cannot force the loader to initialize this section at process creation, you can initialize the data the first time that a running program accesses the installed library. To accomplish this, your source code should declare a Boolean variable that will only be false the first time that a running program attempts to access it. When it is false, the installed library can call an initialization routine.

Global symbols in `/lib/userlib.private` should not duplicate names defined in other installed libraries.

In summary, the system initializes static data in the `data$` section once, but after initialization, the static data becomes read-only. All variables in other data sections are initialized to zero *unless* the variable has other data initialization from the source.

> NOTE: If you used a compiler to create the object module in `/lib/userlib.private`, then all global symbols in the installed library are automatically accessible to running programs. However, if you used the binder to create the object module in `/lib/userlib.private`, then by default the global symbols in the installed library are *inaccessible* to running programs. To make these global symbols accessible to running programs, you must use the -MARK or -ALLMARK option when you bind. See Chapter 2 for full details on these important binder options.

4.5 Object Files Installed With the -INLIB Binder Option

As of SR9.5, you can use the binder's -INLIB option to request that certain libraries be *automatically* installed at execution time. This alleviates much of the need for manually issuing the INLIB command prior to executing the program. It is also more flexible and efficient than the user-defined global library. There can only be one user-defined global library on a workstation, and it is installed in *every* new process, whether or not it's needed. This can significantly degrade the performance of process creation. The -INLIB binder option, on the other hand, only installs those libraries that are needed by the program being executed, and there is no real limit on the number of different libraries that can be handled in this way.

See Chapter 2 for details on the -INLIB option.

4.6 Multiple Global Definitions in Installed Libraries

Sometimes, you install an object file that defines global symbols that have already been defined by another active installed library. In such a case, the new definition overrides the old one. The loader reinstates the old definition if the new installed library becomes uninstalled. For example, suppose a global library defines a global symbol called `pas_$write`. If you install an object file that also defines a global symbol called `pas_$write`, then external references to `pas_$write` will be resolved by the user-defined installed library. If you close the shell in which you installed the object file, then external references to `pas_$write` will be resolved by the global library.

In order to ensure reliable program execution, the loader does not re-evaluate external references resolved at runtime if you install a new installed library while the program is executing. In other words, once a running program accesses a particular global symbol from an installed library, then that symbol cannot be overridden while the program is running. For example, suppose you install object file `j1` as an installed library. Further suppose that you are executing an object file which makes an external reference to symbol `lunar`, and that `j1` defines `lunar` as a global symbol. Also, assume that after your object file accesses `lunar`, you install object file `j2` (which defines `lunar` as a global symbol) as an installed library. During the execution of your object file, the loader will always access `j1`'s `lunar` rather than `j2`'s `lunar`. When your object file stops running, `j2`'s `lunar` will override `j1`'s `lunar` for future program execution.

To reduce the possibility of multiple global definitions, all DOMAIN symbols contain the phrase "_$" (e.g., `pas_$write`, `stream_$get_rec`). To avoid accidentally duplicating a global symbol in a DOMAIN installed library, make sure your symbol names don't include "_$".

Appendix A

Binder Error and Warning Messages

This appendix contains a listing of the errors and warning messages that you may encounter during binding. Each message is classified as either an error or a warning. Warning–level messages indicate conditions that do not prevent the binder from producing an output file. However, warning–level messages may mean that the file's contents are not what you expect. Error–level messages are fatal conditions that prevent the binder from producing an output file.

Attempt to respecify start addr (warning)

More than one input object module specified a start address. Therefore, the binder sets the start address of the output object module to the first possible start address encountered. For example, suppose that object modules `c.bin` and `e.bin` each define a start address. If you issue the following command line:

```
$ bind a.bin b.bin c.bin d.bin e.bin -binary lev1
```

then the binder sets the start address of `lev1` to the start address of `c.bin`. (See Section 3.4 for details on start addresses.)

Bad obj: Duplicate ID number (error)

The input object module has been corrupted. You should recompile the source code to create a new object file. If the error persists, contact your software support representative.

Bad obj: Missing ID number (error)

(See note under 'Bad obj: Duplicate ID number'.)

Bad obj: No global base for rel (error)

(See note under 'Bad obj: Duplicate ID number'.)

Bad obj: No section base for rel (error)

(See note under 'Bad obj: Duplicate ID number'.)

Bad obj: No text for reloc rcrd
(error)

(See note under 'Bad obj: Duplicate ID number'.)

Bad obj: Reloc of odd addr
(error)

(See note under 'Bad obj: Duplicate ID number'.)

Bad obj: Reloc outside text
(error)

(See note under 'Bad obj: Duplicate ID number'.)

Bad obj: Text overflow section
(error)

(See note under 'Bad obj: Duplicate ID number'.)

Binary file already open
(warning)

You specified the –BINARY option more than once in the same command. The binder writes the output object module to the file specified as an argument to the first –BINARY option.

Binary file name cannot start with "–"
(error)

The keyword –BINARY must be followed by a pathname; however, you have mistakenly followed –BINARY with an option.

Cannot close binary output
(error)

The binder cannot close the file you specified with the –BINARY option. This error probably indicates that the output file is unusable and that you should re–execute the bind command to create another output file.

Cannot close input file
(error)

The binder could not close one of the input object files. The probable cause of this error is some sort of network problem. The output object module created by the binder is probably usable.

Cannot close map file
(warning)

You used the –MAP option and tried to redirect standard output to a file, but the operating system could not close this file. Possibly there were network problems when you executed the binder.

Cannot open XREF output file
(warning)

You used the –XREF option and tried to redirect standard output to a file, but the operating system could not open this file. Possibly there were network problems when you executed the binder.

Cannot open file
(error)

This pathname exists, but the operating system cannot open it. Possibly there were network problems when you executed the binder.

Conflicting object system types
(error)

The binder prohibits you from specifying input object files having different systypes. For example, suppose that the compiler stamped a.bin with a systype of "sys5", but b.bin has a systype of "bsd4.2". In this case, the following bind command line will trigger the error:

```
bind a.bin b.bin -binary myprog
```

All input object files must be stamped with the same systype. Don't forget that the system always stamps an object file with a systype even if you didn't specify one. For more information on systype, see the "-SYSTYPE" listing of Chapter 2.

Could not open Binary output
(error)

The filename you specified after the -BINARY option exists, but the operating system cannot open it. Possibly, the file was already open or perhaps the operating system could not delete the old .BAK file because of improper ACLs.

File not found
(warning in interactive mode)
(error in noninteractive mode)

The operating system could not find the specified file. Perhaps you misspelled a pathname, or perhaps network problems prevented the operating system from finding the file.

File skipped - not an object module or a library
(error)

The binder was expecting a file containing either an object module or a library file. However, the file you specified was neither.

Global not defined
(error)

The global symbol you specified as an argument to -MARK or -UNMARK has not been defined yet by an input object module. To correct this error, put the option after an object module that defines it.

Inquire about STDIN
(warning)

The binder made an operating system inquire call, but the operating system detected an error. If the problem persists after recompiling your source files, you should contact your software support representative.

Invalid alignment type for -ALIGN command
(warning)

You specified something other than LONG, QUAD, or PAGE as an argument to the -ALIGN option. Therefore, by default, the binder will align the section on a LONG boundary.

Invalid global name
(error)

You tried to specify a global symbol as an argument to –MARK or –UNMARK, but the name you entered contained some illegal characters.

Invalid module name
(error)

You tried to specify an object module as an argument to –INCLUDE or –MODULE, but the argument you entered contained some illegal characters.

Invalid section name
(error)

You tried to specify a section as an argument to –ALIGN, –LOOKSECTION, –NOLOOKSECTION, –MARKSECTION, –UNMARKSECTION, or –READONLYSECTION, but the name you entered contained some illegal characters.

Invalid start address ignored
(warning)

The binder encountered a possible start address in one of the input object modules that referred to an unknown section. This warning could indicate a compiler error or that one of the input object files has been corrupted.

Invalid system type
(error)

The system name that you requested when you used the –SYSTYPE option was not a valid name or was entered incorrectly. Enter a valid system name. Also, correct any format errors or typos.

Last file known is not a library file, no include done
(error)

The pathname that most closely precedes the –INCLUDE option was not a library file. To correct this error, just change the order of your binder command so that –INCLUDE comes after the library file it refers to.

Library object not an object module
(error)

You used the –INCLUDE option to name an object module from a library, but one of two things went wrong. Either the object module you specified was not actually stored in the library, or there was an object module with this name, but it has somehow become corrupted.

Mixed overlay/concat allocation
(warning)

Your input object modules defined two sections with the same name, but one of these sections had the overlay attribute and the other section had the concatenated attribute. For consistency, the binder assumes that both sections have the overlay attribute.

Mixed R/O and R/W in section
(error)

Two input object modules defined a section with the same name. However, one of these sections had the read–only attribute and the other had the read/write attribute.

Module to include cannot be found in library
(warning)

The object module you specified as an argument to -INCLUDE is not stored in the library file preceding the option. Therefore, the binder ignores this -INCLUDE option.

Multiple resolutions are possible for implicitly resolved symbol
(error)

The named global exists in more than one module within the library. The binder reports this error if you use the -MULTIRES option.

Multiply defined global
(warning)

Two object modules are both trying to define a global symbol with the same name. The binder takes the first one it encounters.

No alignment type for -ALIGN command
(warning)

You forgot to specify LONG, QUAD, or PAGE as an argument to the -ALIGN option. Therefore, by default, the binder aligns the section on a LONG boundary.

No global name to mark
(warning)

You forgot to specify a global symbol as an argument to the -MARK option.

No global to unmark
(warning)

You forgot to specify a global symbol as an argument to the -UNMARK option.

No input
(fatal error)

You supplied no input object files. Therefore, the binder won't generate any error messages, warning messages, or map files.

No input provided to XREF
(warning)

The binder found no sections or global symbols to cross-reference. Perhaps you mistakenly placed the -XREF option at the end of the binder command. -XREF only affects the object files and library files that come after it in the command.

No module name to include
(warning)

You forgot to specify an argument (either an object module name or the keyword -ALL) to the -INCLUDE option.

No name for Binary output
(warning)

You forgot to specify an argument (a pathname) to the -BINARY option.

No name for -MODULE command
(error)

You forgot to specify an argument (the name of the output object module) to the -MODULE option.

No section name for -ALIGN command
(warning)

You forgot to specify a section name as an argument to the -ALIGN command.

No section name for LOOKS
(warning)

You forgot to specify an argument (either a section name or the keyword -ALL) to the -LOOKSECTION option.

No section name for MARKS
(warning)

You forgot to specify an argument (either a section name or the keyword -ALL) to the -MARKSECTION option.

No section name for -NOLOOKS
(error)

You forgot to specify an argument (the name of a section or the keyword -ALL) to the -NOLOOKSECTION option.

No section name for READONLY
(warning)

You forgot to specify an argument (a section name) to the -READONLYSECTION option.

No section name to UNMARK
(warning)

You forgot to specify an argument (either a section name or the keyword -ALL) to the -UNMARKSECTION option.

Not all globals were resolved
(error)

You used the -ALLRES option and forgot to include a module in the bind command line. -ALLRES causes the binder to exit in an error if it finds an undefined global.

Relocation record for section changed to R/O
(error)

The section you specified as an argument to -READONLYSECTION contains relocation information and must, therefore, have the read/write attribute. In other words, you should not attempt to change the attributes of this section. Relocation information consists of addresses to other external objects which must be adjusted by the loader at runtime. You can only make a section read-only if it contains compile-time constants.

Section is already read-only
(error)

You tried to use the -READONLYSECTION option, but the section you specified as an argument already has the read-only attribute. You can only specify as an argument a section with the read/write attribute.

Section must be overlay
(error)

You specified a section as an argument to -LOOKSECTION, -NOLOOKSECTION, -MARKSECTION, -UNMARKSECTION, or -READONLYSECTION, but this section has the concatenated attribute. You can only specify a section that has the overlay attribute.

Section must be read/write
(error)

You specified a section as an argument to -LOOKSECTION, -NOLOOKSECTION, -MARKSECTION, -UNMARKSECTION, or -READONLYSECTION, but this section has the read-only attribute. You can only specify a section that has the read/write attribute.

Section not data
(error)

You specified a section as an argument to -LOOKSECTION, -NOLOOKSECTION, -MARKSECTION, -UNMARKSECTION, or -READONLYSECTION, but this section has the code attribute. You can only specify a section that has the data attribute.

Section not defined
(error)

You specified a section as an argument to -LOOKSECTION, -NOLOOKSECTION, -MARKSECTION, -UNMARKSECTION, or -READONLYSECTION, but no input object module has yet defined this section. Try putting the option later in the binder command line. If that doesn't work, make sure you have entered all the necessary object modules.

Section not defined for -ALIGN command
(error)

You specified a section as an argument to -ALIGN, but no input object module has yet defined this section. Try putting -ALIGN later in the binder command line. If that doesn't work, make sure you have entered all the necessary object modules.

Section table overflow
(error)

The binder attempted to create more than 2,048 sections in the output object module. If you are programming in the C language, note that the compiler assigns each global variable to a separate section. Thus, you may have to reduce the number of global variables in your C source code.

Too many sections in input file
(error)

One of your input object modules has more than 2,048 sections. If you are programming in the C language, note that the compiler assigns each global variable to a separate section. Thus, you may have to reduce the number of global variables in your C source code.

Unknown command ignored
(warning in interactive mode)
(error in noninteractive mode)

You specified an option that the binder doesn't recognize.

Wrong version of object format
(error)

One of your input object modules has an invalid format. Possibly, you are binding with an earlier version of the binder, or possibly you inadvertently modified the input object module.

Wrong version of library format
(error)

One of your input library files has an invalid format. Possibly, you are binding with an earlier version of the binder, or possibly you inadvertently modified the input library.

Wrong version of library object module format
(error)

One of your input library files has an invalid format. Possibly, you are binding with an earlier version of the binder, or possibly you inadvertently modified the input library.

Appendix B

Librarian Error and Warning Messages

This appendix contains a listing of the errors and warning messages that you may encounter while using the librarian. Each message is classified as either an error or a warning.

An error message indicates that the librarian could not perform the requested operation or that some error condition arose while the librarian was trying to perform the operation. In either case, the result is probably an unusable library file.

A warning message indicates that one of the following is true:

- The librarian could perform the requested operation, but the contents of the library file may not be what you were expecting.
- The librarian could not perform the operation, but the library file was not corrupted. Therefore, you can issue a corrected command on the original library file.

Here now is a list of all the error and warning messages produced by the librarian:

Cannot close library output (error)

The librarian encountered an error when it tried to close the library file. This error sometimes indicates that the library has been corrupted; therefore, you should try to recreate the library, if possible.

Cannot close map file (warning)

The map file is the file that the librarian generates in response to the –LIST option. This warning indicates that the librarian encountered a problem when it tried to close this map file. This warning has no affect on the library itself.

Cannot close object output file (error)

You tried to use the –OUTPUT option to copy an extracted object module to an output file, but the librarian could not close the output file. Therefore, the output file is unusable. You have to delete it and try the librarian command over again.

Cannot open file (error)

You specified an object file or library file to be added to the library, but the librarian could not open the specified file. Operating system or network problems are responsible for this error. This error will not corrupt the library file and the librarian will probably have correctly executed everything preceding the error.

Cannot open file, no update done
(error)

You specified a file as an argument to -REPLACE, but the librarian could not open this file. This error will not corrupt the library file. The librarian will have processed commands up until this point.

Could not open library file
(error)

The librarian could not open the named library file. Perhaps this file was being used by some other process. Also, it is possible that there was no disk space or virtual address space available. Or, perhaps there was a network problem when you tried to open the library file. Probably, the library file is uncorrupted.

Could not open object output file, no extract done
(error)

When you used the -OUTPUT option, you specified a pathname that the librarian could not open. Perhaps the error was caused by network problems. The output library file will probably not be corrupted by this error.

CREATE option must be followed by new library pathname
(warning)

You entered the command LBR -CREATE, but you did not specify the pathname of the library to be created. The pathname must be on the same line as -CREATE.

CREATE option specified but named file already exists, can't create
(error)

The librarian interprets the first character string after -CREATE as the filename of the new library. The librarian signals this error if you've entered a filename that already exists. This ensures that you don't overwrite an existing file. Usually, you get this error when you type in the names of the contributing object files and forget to enter the name of the library. This error will not change the existing library file in any way.

File not found
(warning)

You specified that object modules from a certain file should be added to the library file, but the librarian could not find this file. Perhaps you misspelled the filename, or perhaps network problems prevented the librarian from accessing the file.

File not found, no update done
(warning)

You specified a file with the -REPLACE option, but the librarian could not find this file. Perhaps you misspelled the filename, or perhaps network problems prevented the librarian from accessing it.

File specified is not a valid library file
(error)

You specified a file immediately after -UPDATE, but this file does not contain a valid library. Remember that a library is a file created by the librarian. The librarian will not alter the specified file.

Invalid module name, no delete done
(warning)

You entered a module name after –DELETE that does not follow the syntax rules for valid module names. Perhaps it begins with a digit or contains invalid characters.

Invalid module name, no extract done
(warning)

You entered a module name after –EXTRACT that does not follow the syntax rules for valid module names. Perhaps it begins with a digit or contains invalid characters.

Module already exists and replacement was not specified, old module kept
(warning)

You tried to add a module to a library (as opposed to trying to replace the module with the –REPLACE option), but the named module already exists in the library file. Instead of trying to add this object module to the library file, you should try to replace it with the –REPLACE option.

Module does not exist in library, no delete done
(warning)

When you used the –DELETE option, you specified an object module that was not part of the library file. (Note that the librarian is case–sensitive to object module names.) To get a listing of the names of all object modules in the library file, use the –LIST option.

Module name does not exist in library, no extract done
(warning)

When you used the –EXTRACT option, you specified an object module that was not part of the library file. To get a listing of the names of all object modules in the library file, use the –LIST option. Remember that the librarian is case–sensitive to object module names.

Module name is not between 1 and 32 characters in length, no delete done
(warning)

You must supply a module name immediately after the –DELETE option, and that name must be less than 33 characters in length. You probably forgot to specify an object module, or if you did specify an object module, you may have misspelled it.

Module name is not between 1 and 32 characters in length, no extract done
(warning)

You must supply the name of an object module immediately after –EXTRACT, and that name must be less than 33 characters in length. You probably forgot to specify an object module, or if you did specify an object module, you probably misspelled it.

No library specified, no add done
(error)

You forgot to specify –CREATE or –UPDATE. Therefore, the librarian will not be able to perform your request to add new object modules.

No library specified, no replace done
(error)

You forgot to specify –CREATE or –UPDATE. Therefore, the librarian will not be able to perform your request to replace object modules.

No path name specified, no replace done
(warning)

You forgot to put a pathname immediately after -REPLACE. The pathname must be on the same line as the -REPLACE.

Object found in library is not a valid object module, no add done
(warning)

You specified a library file to be added to an existing library, but the library file you wanted to add contains one or more invalid object modules. Perhaps you entered the wrong filename.

Object found was not a valid object module or library, no add done
(warning)

You tried to add object module(s) to a library file, but you didn't enter the name of a valid library or object file. Probably, you entered the wrong filename; for example, you entered `my_prog` instead of `my_prog.bin`.

Object found was not a valid program or library module, no replace done
(warning)

You specified a file after -REPLACE, but this file is neither a valid object file (a compiled program) nor a valid library (a special file created by the librarian). Perhaps you misspelled the pathname.

Open error or named file already exists for output file, no extract done
(error)

There are two possible causes for this error. Perhaps the pathname you specified as an argument to the -OUTPUT (-O) option already exists. (You must specify a pathname that does not currently exist.) The existing file probably will not be corrupted by this error. Another possibility is that you specified a file that did not exist, but the binder could not create it. Perhaps the disk is full, or there are network problems, or the pathname you specified was illegal.

-OUTPUT must be followed by a single valid pathname, no extract done
(warning)

When you used the -OUTPUT option, you forgot to put a single valid pathname immediately after -OUTPUT. This pathname must be on the same line as -OUTPUT.

Previous CREATE option specified, this create option ignored
(warning)

You entered -CREATE more than once in the same LBR command. You cannot create or update more than one library file during a single execution of the librarian. If you enter -CREATE twice, the librarian ignores any filename that comes immediately after the second -CREATE.

Previous CREATE option specified, update not allowed
(error)

You entered -UPDATE in a command containing a previous valid -CREATE option. You cannot update a library in the same command in which you create a library. The librarian executes everything in the command up until the -UPDATE (at which point, it aborts execution). This error will not corrupt the library file.

Previous UPDATE option specified, CREATE not allowed
(error)

You entered -CREATE in a command that contains a previous valid -UPDATE command. A librarian command cannot contain both -CREATE and -UPDATE. If this is the only error, then the librarian probably correctly executed everything up until the -CREATE.

Previous UPDATE option specified, this update option ignored
(warning)

You entered -UPDATE more than once. If this is the only error, then the librarian probably executed everything up until the second -UPDATE.

-REPLACE is followed by an option instead of a pathname, no replace done.
-REPLACE is followed by an option instead of a pathname, option ignored.
(warning)

This double line warning message indicates that the argument after -REPLACE begins with a hyphen (-) and thus cannot be a pathname. (The librarian assumes that arguments beginning with a hyphen are options.) The librarian performs neither the replace nor the option immediately after it. Probably, you forgot to specify a pathname after -REPLACE, or you accidentally put a hyphen before the pathname argument.

Replace of module which is not in library, module is added
(warning)

You used the -REPLACE option, but the named object module does not exist in the library. This is a harmless warning that the librarian issues to inform you that a replace was unnecessary (an add would have sufficed). This lets you double check that you really gave the correct filename.

Unknown Command Ignored
(warning)

You entered a string of characters preceded by a hyphen (-) somewhere in the command line, but the characters do not represent a valid librarian option. Perhaps you misspelled a librarian option, or perhaps you accidentally put a hyphen in front of a filename. To let you know where you went wrong, the librarian prints the faulty string of characters just after this warning.

UPDATE option must be followed by new library pathname
(warning)

You entered the command LBR -UPDATE, but you did not specify the pathname of the library file to be updated. You must enter the pathname on the same line as -UPDATE.

UPDATE option specified but named file does not exists, can't update
(error)

The librarian interprets the first character string after -UPDATE as the filename of an existing library. The librarian signals this error if you've entered a pathname that doesn't exist. Usually, you get this error when you type in the names of some contributing object files and forget to enter the name of the library file they affect.

Appendix C

Section Attributes

This appendix contains a list of the attributes that can characterize a section. This list should help you interpret the maps produced by the librarian –LIST option or the binder –MAP option. The boldfaced portion of the attribute is the abbreviation that you see in the librarian and binder maps. For example, the expression "Abs" appearing in a binder map refers to the Absolute attribute.

Absolute Attribute

A section may or may not have the absolute attribute. The absolute attribute tells the loader to begin this section at a fixed virtual address. If you are programming in a high–level language, you have no control over this attribute.

Data and **Instr**uction Attributes

A section must have either the data attribute or the instruction attribute. The data attribute means that the section contains program data only. The instruction attribute means that the section contains machine code instructions only. Users cannot control these attributes.

Installed (or MARKSECTION) Attribute

A section may or may not have the installed attribute (also called the MARKSECTION attribute). If an installed library contains a section with the installed attribute, then that section can share data with a section of the same name in an installed library or executing noninstalled object file. You control the installed attribute with the –MARKSECTION and –NOMARKSECTION binder options described in Chapter 2.

Long–Aligned, **Quad–Aligned**, and **Page–Aligned** Attributes

A section must have the long–aligned attribute, the quad–aligned attribute, or the page–aligned attribute. A long–aligned attribute means that the loader must install the section beginning on a virtual address that is a multiple of four bytes. A quad–aligned attribute means that the loader must install the section beginning on a virtual address that is a multiple of eight bytes. A page–aligned attribute means that the loader must install the section beginning on a virtual address that is a multiple of 1,024 bytes. You can control these attributes through the –ALIGN binder option described in Chapter 2.

Look_installed (or LOOKSECTION) Attribute

A section may or may not have the look_installed attribute (also called the LOOKSECTION attribute). At runtime, a section with the look_installed attribute can share data with a section of the same name in an installed library. You control the look_installed attribute with the –LOOKSECTION and –NOLOOKSECTION binder options described in Chapter 2.

Overlay and **Concat**enated Attributes, and the **Mixed** Message

A section must have either the overlay attribute or the concatenated attribute. Components of a section with the overlay attribute share the same address space at runtime. Components of a section with the concatenated attribute do not share the same address space at runtime. Instead, they are placed one after another. "Mixed" is not an attribute, but merely indicates that an attempt was made to combine overlay and concatenated components in the same section. When this occurs, the binder or librarian assumes that the section has the overlay attribute. Users have indirect control of these attributes through source code.

Read–Only and **Read/Write** Attributes

A section has either the read–only attribute or the read/write attribute. The abbreviation for the read–only attribute is "R/O". If "R/O" does not appear in the list of attributes, it means that this section has the read/write attribute. A section with the read–only attribute is write–protected, and a section with the read/write attribute is not write–protected. Sections with the read–only attribute reduce system overhead at runtime because the operating system doesn't have to copy the sections out to disk as part of its virtual memory operations. Users have some control of these attributes through the –READONLYSECTION binder option described in Chapter 2.

Zero Attribute

A section may or may not have the zero attribute. If a section has the zero attribute, then the loader sets all of the section's bytes to zero at runtime. A section with the zero attribute must also have the read/write attribute. If you are programming in FORTRAN, you can control this attribute with the –ZERO compiler option. If you are programming in other languages, you have no direct control over this attribute.

Appendix D

Sample Program Development

This appendix provides a basic tutorial for developing programs on the DOMAIN system in FORTRAN, Pascal, or C.

D.1 Sample Source Code

To help demonstrate program development, we've written a simple sample program in FORTRAN, Pascal, and C. We divided the FORTRAN and C programs into four files each and the Pascal program into five files. One file in each language contains the main program and the other files contain subprograms. This arrangement provides a lot of flexibility for developing the program.

The programs printed in this appendix are available on-line. (See the release notes for Pascal, FORTRAN, or C for details on accessing them.)

D.2 Sample FORTRAN Program

Consider a sample FORTRAN program consisting of one "main" program and three subprograms. The main program is stored in file `geoshapes.ftn` and is shown in Figure D-1. The three subprograms are stored in files `math1.ftn`, `math2.ftn`, and `math3.ftn` and are shown in Figures D-2, D-3, and D-4.

```
      INTEGER*2  CHOICE
      REAL       HEIGHT, BASE, AREA, LENGTH, RADIUS
C     EXTERNAL TRIANGLE, SQUARE, CIRCLE

      WRITE (*,*) 'To find the area of a'
      WRITE (*,*) '  triangle (enter 1) '
      WRITE (*,*) '  square   (enter 2) '
      WRITE (*,*) '  circle   (enter 3) '
      WRITE (*,90)
      READ   *,   CHOICE
      GOTO (1, 2, 3) CHOICE


C   TRIANGLE SECTION
1     WRITE (*,100)
      READ   *, HEIGHT, BASE
      CALL TRIANGLE(HEIGHT, BASE, AREA)
      GOTO 200

C   SQUARE SECTION
2     WRITE (*,110)
      READ   *, LENGTH
      CALL SQUARE(LENGTH, AREA)

C   CIRCLE SECTION
3     WRITE (*,120)
      READ *,RADIUS
      CALL CIRCLE(RADIUS, AREA)

C   RESULTS SECTION
200   WRITE (*,80) AREA


80    FORMAT('THE AREA IS ', F7.2)
90    FORMAT('What is your choice -- ', $)
100   FORMAT(' Enter the height and base of the triangle -- ', $)
110   FORMAT(' Enter the length of one side of the square   -- ', $)
120   FORMAT(' Enter the radius of the circle -- ', $)
      END
```

Figure D-1. Code Stored in File GEOSHAPES.FTN

```
      SUBROUTINE TRIANGLE(HEIGHT, BASE, AREA)
      REAL HEIGHT, BASE, AREA

      AREA = .5 * HEIGHT * BASE
      END
```

Figure D-2. Code Stored in File MATH1.FTN

```
      SUBROUTINE SQUARE(LENGTH, AREA)
      REAL LENGTH, AREA

      AREA = LENGTH * LENGTH
      END
```

Figure D-3. Code Stored in File MATH2.FTN

```
      SUBROUTINE CIRCLE(RADIUS, AREA)
      REAL RADIUS, AREA

      AREA = 3.14159 * RADIUS * RADIUS
      END
```

Figure D-4. Code Stored in File MATH3.FTN

D.3 Sample Pascal Program

The following figures show a sample Pascal program consisting of one "main" program and three subprograms. The main program is stored in file geoshapes.pas and is shown in Figure D–5. The three subprograms are stored in files math1.pas, math2.pas, and math3.pas and are shown in Figures D–6, D–7, and D–8. We stored the definitions of all external routines inside one file named external_routines.pas, and it is shown in Figure D–9.

```
program geoshapes;
%include 'external_routines.pas'

VAR
     height, base, length, radius, area : real;
     choice : integer;

BEGIN
     writeln('To find the area of a ');
     writeln('  triangle (enter 1) ');
     writeln('  square   (enter 2) ');
     writeln('  circle   (enter 3) ');
     write('What is your choice -- '); readln(choice);

     case choice of
       1 : begin
                write('enter the height and base of the triangle -- ');
                readln(height, base);
                triangle(height, base, area);
           end;
       2 : begin
                write('enter the length of one side of the square -- ');
                readln(length);
                square(length, area);
           end;
       3 : begin
                write('enter the radius of the circle -- ');
                readln(radius);
                circle(radius, area);
           end;
       otherwise  return;
     end;
     writeln('The area is ', area:4:2);
END.
```

Figure D–5. Code Stored in File GEOSHAPES.PAS

```
module math1;
DEFINE triangle;
%include 'external_routines.pas';

PROCEDURE triangle;
BEGIN
   area := (0.5 * height * base);
END;
```

Figure D–6. Code Stored in File MATH1.PAS

```
module math2;
DEFINE square;
%include 'external_routines.pas';

PROCEDURE square;
BEGIN
   area := length * length;
END;
```

Figure D-7. Code Stored in File MATH2.PAS

```
module math3;
DEFINE circle;
%include 'external_routines.pas';

PROCEDURE circle;
CONST
   pi = 3.14;
BEGIN
   area := pi * radius * radius;
END;
```

Figure D-8. Code Stored in File MATH3.PAS

```
procedure triangle(in height : real;
                   in base   : real;
                   out area  : real); extern;
procedure square(in length   : real;
                 out area    : real); extern;
procedure circle(in radius   : real;
                 out area    : real); extern;
```

Figure D-9. Code Stored in File EXTERNAL_ROUTINES.PAS

D.4 Sample C Program

The following figures show a sample C program consisting of one "main" program and three subprograms. The main program is stored in file geoshapes.c and is shown in Figure D-10. The three subprograms are stored in files math1.c, math2.c, and math3.c and are shown in Figures D-11, D-12, and D-13.

```
extern float  triangle();
extern float  square();
extern float  circle();

float  height, base, length, radius, area;
int    choice;

main()
{
     printf("To find the area of a \n");
     printf("  triangle (enter 1) \n");
     printf("  square   (enter 2) \n");
     printf("  circle   (enter 3) \n");
     printf("What is your choice -- "); scanf("%d", &choice);

     switch (choice)
     {
       case 1 :   printf("enter the height and base of the triangle -- ");
                  scanf("%f%f", &height, &base);
                  area = triangle(height, base);
                  break;
       case 2 :   printf("enter the length of one side of the square -- ");
                  scanf("%f", &length);
                  area = square(length);
                  break;
       case 3  :  printf("enter the radius of the circle -- ");
                  scanf("%f", &radius);
                  area = circle(radius);
                  break;
       default :  return;
     }
     printf("The area is %5.2f\n", area);
}
```

Figure D-10. Code Stored in File GEOSHAPES.C

```
float  triangle(height, base)
float  height, base;
{
   return(0.5 * height * base);
}
```

Figure D-11. Code Stored in File MATH1.C

```
float  square(length)
float  length;
{
   return(length * length);
}
```

Figure D-12. Code Stored in File MATH2.C

```
float  circle(radius)
float  radius;
{
#define pi 3.14
   return(pi * radius * radius);
}
```

Figure D-13. Code Stored in File MATH3.C

D.5 Compiling

The next step in program development is to compile the source code. You must compile each file of source code separately by using the commands shown in Table D-1.

Table D-1. Compiling the Source Code

FORTRAN	Pascal	C

`$ ftn geoshapes`
`$ ftn math1`
`$ ftn math2`
`$ ftn math3`
	`$ pas geoshapes`
	`$ pas math1`
	`$ pas math2`
	`$ pas math3`
		`$ cc geoshapes`
		`$ cc math1`
		`$ cc math2`
		`$ cc math3`

Whether you used the FORTRAN compiler, the Pascal compiler, or the C compiler, the results are basically the same. Namely, the compiler creates object files geoshapes.bin, math1.bin, math2.bin, and math3.bin.

D.6 Possible Program Development Paths

After compiling the source code, there are many ways in which you can create an executable object file. We illustrate several methods in this section.

D.6.1 Path 1: Binding

The most straightforward method for creating an executable object file is to bind all four object files into one executable output object file as follows:

```
$ bind geoshapes.bin math1.bin math2.bin math3.bin -binary geo
```

To execute the output object file, you merely enter its name as a command, for example:

```
$ geo
```

D.6.2 Path 2: Creating a Library File, Then Binding

There is nothing wrong with the development shown in path 1; however, it may be advantageous to build a library file instead as follows:

```
$ lbr -create mathlib  math1.bin math2.bin math3.bin
```

Then you can bind the main routine to the library as follows:

```
$ bind geoshapes.bin mathlib -binary geo
```

And execute the program as you would execute any program, for example:

```
$ geo
```

Why do it this way? Because it might be very helpful for future program development to build a library file of related mathematical functions. The big advantage of a library file is that you have many object files at your disposal, but the binder will only gather the object files required for the binding. Thus, your output object file won't contain any excess code.

D.6.3 Path 3: Using the INLIB Utility

Here, we will use the INLIB utility to build an installed library. First, we will use the binder to build a nonexecutable object file named to_be_installed as follows:

```
$ bind -allmark math1.bin math2.bin math3.bin -binary to_be_installed
```

Notice that we had to use the -ALLMARK binder option to make available the global symbols in math1.bin, math2.bin, and math3.bin.

Now, we will use the INLIB utility to install this object file as follows:

```
$ inlib to_be_installed
```

And finally, we can run the main program without ever running it through the binder as follows:

```
$ geoshapes.bin
```

The loader will match the unresolved external symbols of `geoshapes.bin` with the global symbols in `math1.bin`, `math2.bin`, and `math3.bin`.

Index

The letter *f* means "and the following page"; the letters *ff* mean "and the following pages". Symbols are listed at the beginning of the index. Entries in color indicates procedural information.

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *DOMAIN Binder and Language Reference*
Order No.: 004977 Revision: 02 Date of Publication: February, 1987

What type of user are you?

_____ System programmer; language _______________
_____ Applications programmer; language _______________
_____ System maintenance person _____ Manager/Professional
_____ System Administrator _____ Technical Professional
_____ Student Programmer _____ Novice
_____ Other

How often do you use the DOMAIN system? _______________

What parts of the manual are especially useful for the job you are doing?

____

What additional information would you like the manual to include?

____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)

____

____
Your Name Date

____
Organization

____
Street Address

____
City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *DOMAIN Binder and Language Reference*
Order No.: 004977 Revision: 02 Date of Publication: February, 1987

What type of user are you?

_____	System programmer; language ____________		
_____	Applications programmer; language ____________		
_____	System maintenance person	_____	Manager/Professional
_____	System Administrator	_____	Technical Professional
_____	Student Programmer	_____	Novice
_____	Other		

How often do you use the DOMAIN system? ____________________

What parts of the manual are especially useful for the job you are doing?

____

What additional information would you like the manual to include?

____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)

____

____
Your Name Date

____
Organization

____
Street Address

____
City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD