Domain/OS Programming Environment Reference

apollo

Domain/OS Programming Environment Reference

Order No. 011010-A01

First Printing: July 1988
Latest Printing: October 1989

10 9 8 7 6 5 4 3 2 1

Preface

Domain/OS Programming Environment Reference describes some of the programming tools and capabilities available under Domain/OS.

We've organized this manual as follows:

Chapter 1	Provides an overview of the Programming Environment provided by Domain/OS.
Chapter 2	Introduces some of the programming tools available under Domain/OS.
Chapter 3	Describes the format of object files and the function of the loader under Domain/OS.
Chapter 4	Describes installed libraries, a feature of Domain/OS.
Chapter 5	Describes the procedure calling conventions used by 680x0 and *PRISM* workstations under Domain/OS.
Chapter 6	Details the COFF (Common Object File Format) used for object files under Domain/OS.
Chapter 7	Describes the use of **bind**, a linker.
Chapter 8	Describes the use of **lbr**, the object file librarian.

Summary of Technical Changes

Domain/OS Programming Environment Reference contains technical changes that have been made since its last printing. The changes include addition, subtraction, and augmentation of chapters. Specifically, Chapters 2 and 3 have been added, and all the chapters concerning tools used in the BSD or SysV UNIX environments have been moved to one of the following books:

- *Domain/OS BSD UNIX User's Manual* (017271)
- *Domain/OS BSD UNIX Programmer's Manual* (017272)
- *Domain/OS SysV User's Guide* (017269)
- *Domain/OS SysV Programmer's Guide* (017270)

In addition, the following chapters have been significantly modified:

Chapter 5 Contains additional material about *PRISM* workstations under Domain/OS.

Chapter 6 Includes updated details of the COFF (Common Object File Format) used for object files under Domain/OS.

Related Manuals

The file **/install/doc/apollo/os.v.***current Domain/OS release***__manuals** lists current titles and revisions for all available manuals.

For example, at SR10.0 refer to **/install/doc/apollo/os.v.10.0__manuals** to check that you are using the correct version of manuals. You may also want to use this file to check that you have ordered all of the manuals that you need.

(If you are using the Aegis environment, you can access the same information through the Help system by typing **help manuals**.)

Refer to the *Domain Documentation Quick Reference* (002685) and the *Domain Documentation Master Index* (011242) for a complete list of related documents. For more information on the programming tools and environments available under Domain/OS, refer to the following documents:

- *Using Your Aegis Environment* (011021)
- *Using Your BSD Environment* (011020)

- *Using Your SysV Environment* (011022)
- *Aegis Command Reference* (002547)
- *BSD Command Reference* (005800)
- *Domain/OS BSD UNIX User's Manual* (017271)
- *Domain/OS BSD UNIX Programmer's Manual* (017272)
- *BSD Programmer's Reference* (005801)
- *SysV Command Reference* (005798)
- *Domain/OS SysV User's Guide* (017269)
- *Domain/OS SysV Programmer's Guide* (017270)
- *SysV Programmer's Reference* (005799)

You can order Apollo documentation by calling **1-800-225-5290**. If you are calling from outside the U.S., you can dial **(508) 256-6600** and ask for **Apollo Direct Channel**.

Does This Manual Support Your Software?

This manual was released with software version 10.2. To verify which version of operating system software you are running, type:

```
bldt
```

If you are running Domain/IX on a release of the operating system earlier than SR10.0, then type:

```
/com/bldt
```

If you are using a later version of software than that with which this manual was released, use one of the following ways to check if this manual was revised or if additional manuals exist:

- Read Chapter 3 of the release document that shipped with your product. The release document is online: **/install/doc/apollo/os.v.*x.x*__notes**. Check with your system administrator if you cannot find the release document.
- Telephone **1-800-225-5290**. If you are calling from outside the U.S., dial **(508) 256-6600** and ask for **Apollo Direct Channel**.
- Refer to the lists of manuals described in the preceding section, "Related Manuals."

To determine which of two versions of the same manual is newer, refer to the order number that is printed on the title page. Every order number has a 3-digit suffix; for example, **-A00**. A higher suffix number indicates a more recently released manual. For example, a manual with suffix **-A02** is newer than the same manual with suffix **-A01**.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo Product Reporting (APR) system for comments related to hardware, software, and documentation. By using this formal channel, you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis, BSD, or SysV). Refer to the **mkapr** (make apollo product report) shell command description. You can view the same description online by typing:

man mkapr (in the UNIX shells)

help mkapr (in the Aegis shell)

Alternatively, you may use the Reader's Response Form at the back of this manual to submit comments about the manual.

Documentation Conventions

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

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Pathnames are also in bold. Bold words in text indicate the first use of a new term.
user-supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
sample user input	In interactive examples, information that the user enters appears in color.
`output/source code`	Information that the system displays appears in this `typeface`. Examples of source code also appear in this `typeface`.
[ ]	Square brackets enclose optional items in formats and command descriptions.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/ ^	The notation CTRL/ or ^ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.
. . .	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.
▌	Change bars in the margin indicate technical changes from the last revision of this manual.
——— ⊞ ———	This symbol indicates the end of a chapter or part of a manual.

——— ⊞ ———

Contents

Chapter 1 Domain/OS Features and Environments

Chapter 2 Software Development in Domain/OS

Chapter 3 Object Files in Domain/OS

Chapter 4 Installed Libraries

Chapter 5 Calling Conventions

Chapter 6 COFF: Common Object File Format

Figures

Tables

Chapter 1

Domain/OS Features and Environments

This book is a general introduction to program development on the Domain system.

This chapter is an introduction to the features of Domain/OS, the Domain operating system as of SR10, and its three operating environments. Some of the Domain/OS tools and features traditionally associated with the UNIX environment are documented in the *Domain/OS BSD UNIX User's Manual*, *Domain/OS BSD UNIX Programmer's Manual*, *Domain/OS SysV User's Guide*, and the *Domain/OS SysV Programmer's Guide*.

Domain/OS features let each Apollo workstation easily and efficiently export its services to the rest of the network, and import the services of other nodes. Domain/OS services include file system access and concurrency control, computing capability, peripheral access, and system administration functions. Domain/OS protocols link system nodes without compromising their independence, while at the same time offering much higher level services than those available from traditional networks.

1.1 Domain/OS Features

This section is an overview of some of the more important features of Domain/OS.

1.1.1 Multiple Environments

Domain/OS combines the Aegis operating environment and the UNIX operating environments BSD and SysV. You can use Domain/OS as any one of these environments, or as a combination of all three. Section 1.2 describes this feature in detail.

1.1.2 Multiple Processes

In Domain/OS, each program runs in a unique computing context known as a process. As of SR10.2, you can run as many as 56 simultaneous processes on an MC680x0 based node, and as many as 246 on a Series 10000 node. You can create processes that have pads and windows that let you enter data and view program output. You can also create background processes that run without using the display.

Invoking a program automatically creates a process for that program. You can invoke programs from shells or through Display Manager commands.

Shells are described briefly in Section 1.2.2. The manuals *Using Your BSD Environment*, *Using Your SysV Environment*, and *Using Your Aegis Environment* provide detailed descriptions of shells, the Display Manager, how to invoke programs, and how to create and control processes.

1.1.3 Display Management Systems

The Domain/OS display presents all of its output in windows: sizable, movable, rectangular portions of the screen. To manipulate windows, you can use either of the two window management systems included in Domain/OS: the Display Manager and the X Window System. Both systems allow you to change the sizes and positions of windows. At SR10.2, Domain/OS can accomodate as many as 110 windows (100 pads) on a single node. The Display Manager also allows you to read and edit text files and to create, start, and stop processes. The Display Manager is described in *Using Your BSD Environment*, *Using Your SysV Environment*, and *Using Your Aegis Environment*. The X Window System is described in *Using the X Window System on Apollo Workstations*.

1.1.4 Graphics

Domain/OS provides a flexible graphics environment that supports a wide range of graphics applications. Using the various Domain graphics tools, programs may manipulate workstation displays at the pixel level or use sophisticated three-dimensional graphics routines to display solid objects.

The Graphics Metafiles Resource (GMR) takes full advantage of the high degree of integration offered by all Domain workstations. GMR provides a comprehensive set of 2-D and 3-D graphics structuring, editing, and viewing functions for the creation and display of wireframe and shaded objects.

GMR creates permanent files containing hierarchically structured graphics data. These files can exist on remote nodes and be demand-paged into the virtual memory of the node accessing them for editing or viewing. GMR also supports multiple viewports and automatically rescales to Display Manager windows. See *Programming with 2D Graphics Metafile Resource* (5097), *Domain 2D Graphics Metafile Resource Call Reference* (9793), *Pro-*

gramming with 3D Graphics Metafile Resource (5807), and the *Domain 3D Graphics Metafile Resource Call Reference* (5812) for more information about GMR routines.

The Graphics Primitive Resource (GPR) is a rich set of graphics functions with full text and font capability and a wide range of draw and fill primitives. Graphical input is also supported by this package. Unlike GMR, the GPR routines do not retain an internal representation of the graphics display. See *Programming with Domain Graphics Primitives* (5808) and the *Domain Graphics Primitives Resource Call Reference* (7194) for more information.

The Graphics Service Routines (GSR) provide an interface to allow sophisticated programmers to issue packets of graphics opcodes directly to Domain graphics hardware. Commonly issued calls can be grouped into large packets of opcodes and sent directly to the hardware without checking or procedure call overhead. Like GPR, the GSR do not retain an internal representation of the graphics display. See *Domain Graphics Instruction Set* (9791) and *Programming with Domain Graphics Service Routines* (9797) for more information.

For the creation and maintenance of more portable graphics applications, Domain/OS supports a variety of graphics standards. Domain/PHIGS and Domain/GKS provide support for two popular standards. Domain/PHIGS is an implementation of the PHIGS (Programmer's Hierarchical Interactive Graphics System) ANSI standard, and Domain/GKS is an implementaion of the ISO Graphical Kernel Standard (GKS), at level 2c. See *Programming with Domain/PHIGS* (9701), *Domain/PHIGS Call Reference* (9702), and *Domain/GKS User and Reference Manual* (12077) for more information about the Apollo implementation of these standards. Also, Domain 4014 allows Domain workstations to emulate the Tektronix 4014 graphics terminal, and Domain Core is an implementation of the SIGGRAPH Core standard. See *Domain 4014 User's Guide* (5224) and *Programming with Domain Core Graphics* (1955) for more information.

1.1.5 Distributed File System

Domain/OS enables any node on the network to access any file on the network, unless the owner of the file has protected it. Each node has direct memory access to files on other nodes, so file access across the network is transparent—you don't need to issue any special cross-network access commands or copy a file to your own node in order to access it. Since data copies are unnecessary, disk space isn't wasted with redundant files and accuracy isn't lost as multiple copies become inconsistent.

A concurrency control protocol prevents multiple users from accessing the same file simultaneously.

A network-wide hierarchical directory structure, often called the naming tree, provides the naming scheme for identifying any file anywhere on the network.

For more information on the distributed file system, read about the naming tree in *Using your BSD Environment*, *Using Your SysV Environment*, and *Using Your Aegis Environment*.

1.1.6 Distributed Computation Environment

Domain/OS also enables nodes on the network to use each other's computing power. The Network Computing System (NCS) is a set of tools that allows a programmer to distribute different parts of a computing task to any number of different machines in a heterogeneous network. It supports situations where both data *and* execution are distributed, and where the network mixes vendors, user environments, and hardware. A user may also simply use the **crp** command to run entire programs or scripts on other nodes.

For more information about NCS, see *Managing the NCS Location Broker* (11895) and *Network Computing System Reference* (10200) For more information on the **crp** command, see *Using your Aegis Environment*.

1.1.7 Demand-Paged Virtual Memory

When Domain/OS allocates space for a program, it allocates virtual memory (backed by disk storage) rather than physical memory. This allows you to develop and run programs that are larger than the available physical memory. As the program runs, the system swaps pages of code and data between disk and physical memory, loading the pages it needs and returning to disk the pages that are no longer necessary.

1.1.8 Network-Level Compatibility

All Apollo workstations communicate using network protocols that provide a high level of transparency and services. All Domain hardware and software products are created to be fully compatible at the network level.

1.1.9 Extensible Object Management

Domain/OS employs typed object management. It manipulates each object by calling the operations that are defined for the object's type and implemented by the object's manager. Domain/OS allows you to extend its repertoire of object types by writing new managers. For example, the Open Systems Toolkit enables you to add new I/O objects such as device drivers and file types. Many I/O systems can only add new I/O types by modifying the system source code. In Domain/OS, you can invent a new I/O type and write an independent manager to implement it. The system can communicate with the new type through its manager; no modification of the system source is required.

For more information, see *Using the Open System Toolkit to Extend the Streams Facility* (8863).

1.1.10 Online Help

Online help is available with the Aegis **help** command, the BSD and SysV **man** command, and the **Help** key.

1.2 Domain/OS Environments

Domain/OS combines three previously distinct operating systems: Aegis, BSD, and SysV, which we refer to as **environments** within Domain/OS. This means that you can run Domain/OS as if it were only Aegis, as if it were BSD, as if it were SysV, or you can use features of all three. Any combination of the three environments may exist on a single node.

Domain/OS provides all the utilities, system calls, library calls, and shells associated with the three environments. You can run any combination of shells and utilities, and your Domain/OS programs can use any of the system calls and library calls.

> **NOTE:** All the environments share the kernel operating system functions. What distinguishes one environment from another is simply the shells it supports, its default search path, and the system services associated with it.

1.2.1 System Directories

Aegis, BSD, and SysV each have an associated set of system directories. Aegis traditionally stores its commands in **/com**; BSD stores its commands in **/usr/ucb**, **/bin**, and **/usr/bin**; SysV stores commands in **/bin** and **/usr/bin**. Since Domain/OS contains all three complete environments, it contains three complete sets of system directories, as illustrated in Figure 1-1. Each node has the system directories corresponding to the environment(s) installed on that node: if you install BSD and Aegis, for example, you will have the BSD and Aegis directories; if you install all three environments, you will have all three sets of system directories.

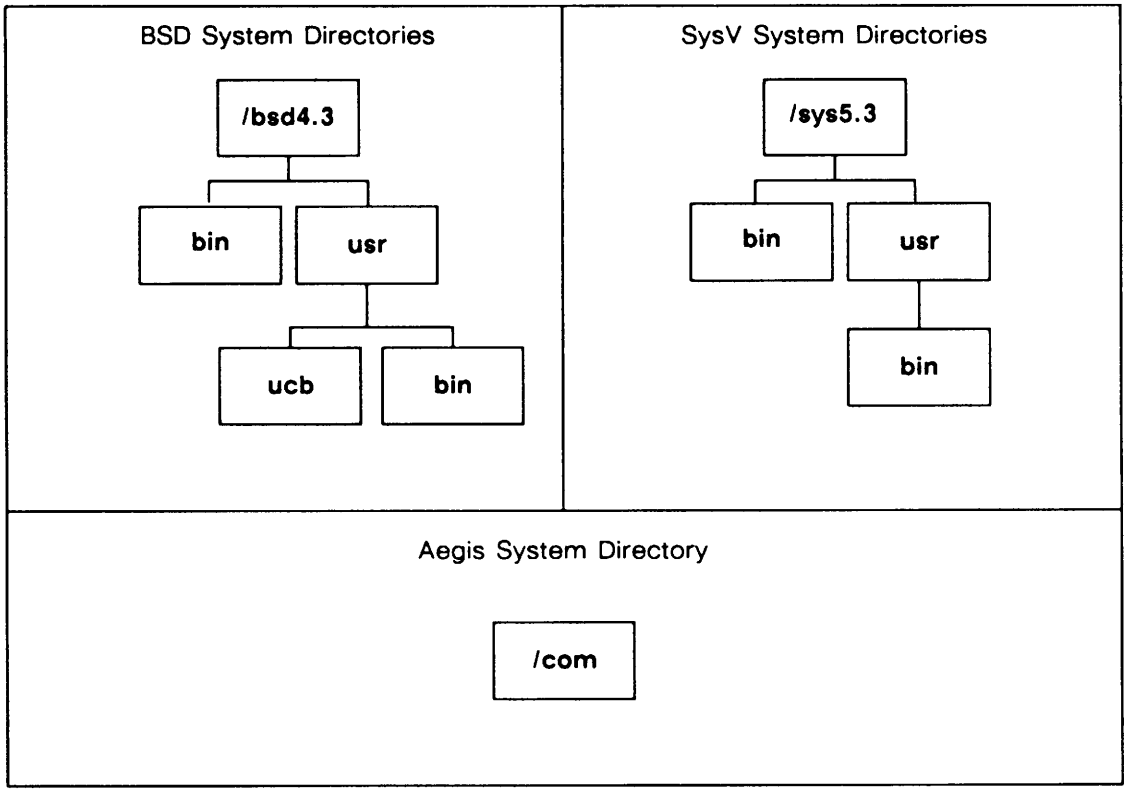


Figure 1-1. System Directories in Domain/OS

Since BSD and SysV both have system directories named **/bin** and **/usr/bin**, we created trees rooted at the directories **/bsd4.3** and **/sys5.3** to hold the two different versions of the identically-named directories. You need not specify these long pathnames, however. You can simply access **/usr/bin**, for example; an environment variable named SYSTYPE determines whether **/usr/bin** points to **/bsd4.3/usr/bin** or **/sys5.3/usr/bin**.

The pathnames **/bin** and **/usr/bin** are actually links to the pathnames **$(SYSTYPE)/bin** and **../$(SYSTYPE)/usr/bin**. If the value of SYSTYPE is **bsd4.3**, therefore, **/bin** is a link to **/bsd4.3/bin**; if SYSTYPE is **sys5.3**, **/bin** is a link to **/sys5.3/bin**.

> **NOTE:** Domain/OS *does* have a directory named **/usr**, which is not a link. Directories such as **/usr/apollo** are part of this **/usr** directory; they aren't sub-directories of **/bsd4.3** and **/sys5.3**.

Within a program or from any shell, you can override the value of SYSTYPE by explicitly naming either the BSD or the SysV tree. For example, you can enter **/bsd4.3/bin/date** to invoke the BSD version of **date**, regardless of the value of SYSTYPE.

The manuals *Using Your Aegis Environment*, *Using Your BSD Environment*, and *Using Your SysV Environment* describe how to set default values for SYSTYPE and how to change the SYSTYPE value in a shell.

If you write a program that depends on a specific value of SYSTYPE, you can use compiler options documented in the language reference manual, or linker options documented online in man pages and help files, to set a **systype** stamp in the object file. At run time, the loader uses the program's **systype** stamp to set the value of SYSTYPE for the program's process. Regardless of the SYSTYPE in the process from which the program is invoked, the SYSTYPE will be correct in the process in which it runs.

1.2.2 Shells

A shell is an interactive program from which you can invoke other programs. You can run any Domain/OS program in any shell. Like all programs, each shell runs in its own process. Like all processes, each shell has its own value of the SYSTYPE environment variable.

The Aegis environment supports only the Aegis shell. If you are running Domain/OS as if it were Aegis alone, you would use only the Aegis shell. The BSD and SysV environments both support Bourne, Korn, and C shells. If you are running Domain/OS as BSD, you would use one or more of the BSD versions of those shell types. If you are running Domain/OS as SysV, you would use one or more of the SysV versions of those three shell types. Note that you can simultaneously run all four types of shells in any combination, if their associated environments have been installed on your node.

The manuals *Using Your Aegis Environment*, *Using Your BSD Environment*, and *Using Your SysV Environment* describe how to create and use shells.

> **NOTE:** The commands to create the three UNIX shells reside in **/bin**. Since **/bin** is a link to either **/sys5.3/bin** or **/bsd4.3/bin**, a command such as **/bin/sh** may create a SysV shell or a BSD shell, depending on the value of SYSTYPE. As always, you can override the value of SYSTYPE by specifying the full pathname. To create a SysV Bourne shell from a BSD Bourne shell, for example, type **/sys5.3/bin/sh**.

1.2.3 Search Paths

The search path specifies where the shell looks for the programs that you invoke.

If you specify a program's full pathname, the shell looks in the directory you specify. For example, if you enter

```
/usr/bin/mail
```

the shell looks in the **/usr/bin** directory for a program named **mail**.

However, if you type only the leafname of the command, the system looks in the directories on the default search path. For example, if you enter

```
mail
```

the shell looks in the directories on the default search path for a program named **mail**.

Each of the four shell types has a specific default search path. The manuals *Using Your Aegis Environment*, *Using Your BSD Environment*, and *Using Your SysV Environment* describe each shell's default search path and how to change it.

> NOTE: To invoke a program that's not on the shell's default search path, enter the program's full pathname. For example, to invoke the Aegis command **date** from a Bourne shell, enter **/com/date**.

1.2.4 System Services

System services are the programmer's interface to the operating system: system calls and library calls that your programs can make. Aegis, BSD, and SysV each have an associated set of system services, although under Domain/OS, a single program can call Aegis services, BSD services, SysV services, or any combination of them. If your programs will be ported to other UNIX systems, of course, you should use only the system services provided by those systems. The Aegis system services are documented in *Programming with Domain/OS Calls* (5506) and *Domain/OS Call Reference (Volumes 1 and 2)* (7196 & 12888) The SysV system services are documented in the *SysV Programmer's Reference*. The BSD system services are documented in the *BSD Programmer's Reference*.

The system calls and library calls for all three environments are always automatically installed with Domain/OS, regardless of how the installation is configured. The header files that define the calls are specific to each compiler language and are optional. Before you compile source code that calls system services, be sure that the appropriate header files are installed on your node. See the relevant compiler documentation for more information.

Aegis system service names are unique, so mixing Aegis calls with BSD or SysV calls presents no problem to programs running in Domain/OS. Many BSD and SysV services, however, have the same names. Typically, these identically named services have different **semantics**, that is, different behaviors.

By default, the system uses the semantics of the environment specified by SYSTYPE. If a program needs to override this default, you can set the **runtype** stamp in the program's object file. The runtype stamp specifies which environment's semantics will be used for system services. You can use compiler options documented in the language reference manuals, or linker options documented in online man pages and help files, to set the **runtype** stamp.

1.2.5 Choosing an Environment

Each of the three Domain/OS environments has strengths and weaknesses. Some programmers find it useful to use features of all three environments, while others prefer to use Domain/OS as if it were a single environment. The advantage of using more than one is that there is a broader range of tools available to you. The disadvantage is that the differences between the environments can inadvertently lead to errors, because UNIX and Aegis environments handle some things differently, such as arguments to commands. If you are already familiar with one environment, you will find it simpler to continue using that environment initially, although eventually you might find it useful to try the other environments because of the wider range of tools that will be available to you. If you are unfamiliar with both UNIX and Aegis, you will find it most helpful to use the environment most commonly used at your site.

If a program is sensitive to its context, if it uses a SysV command, for example, or if it uses system services under BSD that are named the same as SysV services, it may be necessary to specify the runtype and systype stamps in the object file. Otherwise, a program created in one environment will run equally well in any other environment, under any other shell.

Many of the tools and utilities are available in all three environments. See Chapter 2 for a review of some of the programming tools available under Domain/OS.

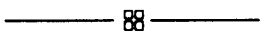
1.2.5.1 Guaranteed Environment

In order to provide a minimum guaranteed UNIX environment so that our third party vendors' installation scripts will run, we provide, in all operating system installations, a small

subset of the SysV environment. Every install, therefore, ensures that the following commands are in the **/sys5.3/bin** directory.

cat	**expr**	**rm**
chgrp	**find**	**rmdir**
chmod	**grep**	**sed**
chown	**id**	**sort**
cp	**ln**	**sum**
cpio	**ls**	**uniq**
cmp	**mkdir**	**tar**
diff	**mv**	**wc**

In addition, the guaranteed UNIX shell, if no other one is available, is the SysV Bourne shell in **/etc/sys_sh**.



Chapter 2

Software Development in Domain/OS

This chapter briefly describes the software development process in Domain/OS and the tools available to the engineer using Domain/OS. In situations where you can choose among several tools to perform a programming task, we provide you with summaries of tools' capabilities to help you make your choice. We also provide references to detailed documentation for each tool.

This chapter is arranged in the following sections:

- Using the Shell as a Prototyping Tool
- Compiling
- Linking
- Using Installed Libraries
- Using Library Files
- Source Code Control and Configuration Management
- Executing Programs
- Debugging
- Analyzing Program Performance
- Using Other Programming Tools
- Domain/OS Programming Tools at a Glance

You should first ensure that the environments and tools of your choice are installed on your workstation. The environments include some of the tools; you must separately order and install others. See your system administrator for information about the environments and tools available at your site.

Figure 2–1 illustrates the interaction of the various utilities. The upcoming chapters take a closer look at each of the program development components on your Domain system.

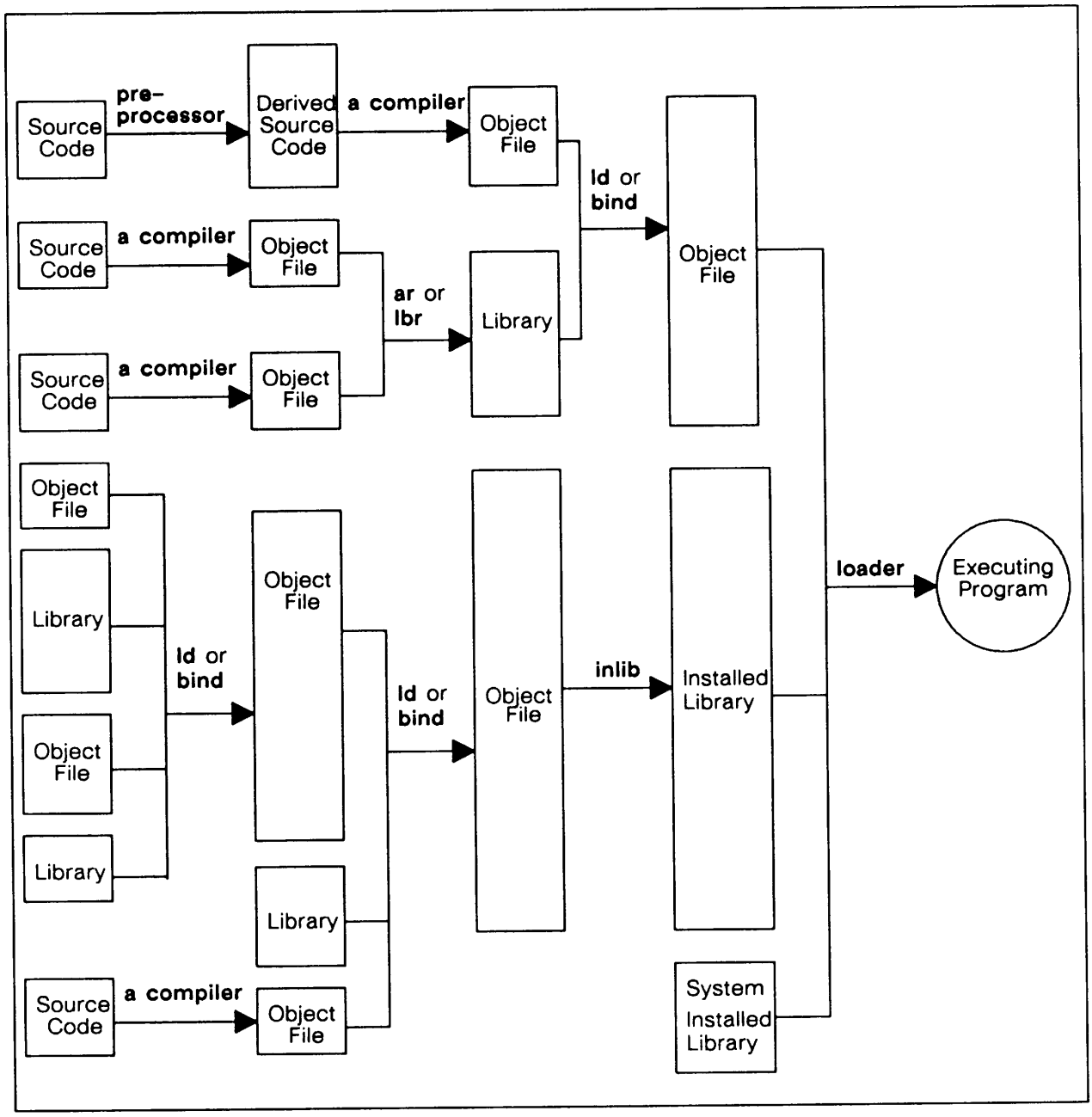


Figure 2–1. The Interaction of Program Development Utilities

2.1 Using the Shell as a Prototyping Tool

Because of a shell's ability to restart processes, direct the flow of control, field interrupts, and redirect input and output, you can consider the shell to be a full-fledged programming language. Programs that use these shell capabilities are known as shell procedures or shell scripts.

Much innovative use of the shell involves stringing together commands to be run under the control of a shell script. The wide range of commands that can be used are documented in *Aegis Command Reference, SysV Command Reference,* and *BSD Command Reference.*

Shell procedures can play an important role in developing prototypes of full-scale applications. While understanding all the nuances of shell programming can be a fairly complex task, getting a shell procedure up and running is far less time-consuming than writing, compiling, and debugging compiled code.

This ability to get a program into production quickly is what makes the shell a valuable tool for program development. Shell programming allows you to "build on the work of others" to the greatest possible degree, since it allows you to piece together major components simply and efficiently. Many times you can develop even large applications using shell procedures. Even if you are initially developing an application as a prototype system for testing purposes rather than putting it into production, you can save much work. With a prototype for testing, you can determine the range of possible user errors—that is not always easy to plan out when an application is being designed. The method of dealing with strange user input can be worked out inexpensively, avoiding large re-coding problems.

For more information on writing shell procedures, you can refer to *Using Your Aegis Environment, Using Your BSD Environment,* and *Using Your SysV Environment.*

2.2 Compiling

All of the Domain compilers accept source code as input and produce **object files** as output. Object files usually have the filename extension **.bin** (Aegis) or **.o** (BSD or SysV). An object file contains machine language code and data in a form that can be used by the linkers, the librarian, or the loader. Chapter 3 describes how Domain/OS manipulates object files.

You can do the following with object files:

- In some cases, you can execute or debug the object file directly.
- You can use one of the linkers to combine several object files to produce a single object file. Chapter 7 describes the Domain linkers.

- You can use one of the librarians to combine several object files to produce a **library file**. Section 2.5 discusses librarians and library files. Chapter 8 describes **lbr**, one of the two librarians available in Domain/OS.

- You can install the object file as an **installed library**. An installed library is a set of subroutines and data items that programs can access at run time. Section 2.4 and Chapter 4 describe installed libraries.

Beginning at SR10.0, the Domain compilers and linkers produce object files in Common Object File Format (COFF), a format used by many UNIX systems. COFF makes it easier for utilities that depend on information in the object file to work on different machines running different versions of the UNIX system. Domain compilers and linkers produce COFF files in all three environments. Chapter 6 details the Apollo implementation of the COFF format.

The compilers generate errors, warnings, and informational messages. An **error** indicates a problem severe enough to prevent the compiler from creating an executable object file. A **warning** is less severe than an error; a warning does not prevent the compiler from creating an executable object file. The warning message tells you about a potential ambiguity in your program for which the compiler believes it can generate the correct code. **Informational messages** are intended to inform you of potential problems in your program.

2.2.1 Cross Compilation

Different Apollo workstations may have different basic instruction sets. An object file which will run on a DN2500 will not run on a DN10000, for example, although it will run on a DN3500. Sometimes you may find it necessary to cross-develop for Apollo workstations: you may be working on a DN4500 and want to produce object code for a Series 10000 workstation, or you may be working on a DN580 and want to produce object code that takes advantage of the special features in a DSP90 workstation. The Domain/OS programming tools are designed with these types of cross-development in mind.

By default, Domain compilers running on 680x0-based workstations generate object code that will run on any 680x0-based workstation. Domain compilers running on Series 10000 workstations generate object code that will run on any Series 10000 workstation. Compiler options, however, allow you to produce code tailored for a specific type of 680x0-based workstation, or to generate Series 10000 code from a 680x0-based workstation, or to produce 680x0-based code from a Series 10000 workstation. Consult the compiler documentation for the appropriate option for the code you're developing.

2.2.2 Compiling for Portability

The Domain/OS operating system has an extensive set of system services such as system calls and library routines. Some of these services are common to many UNIX systems, while others are unique to Domain/OS.

You can use any of these services in your programs, but you should restrict yourself to the standard UNIX services if your programs will be ported to other UNIX systems. Programs that use only the proprietary Domain/OS services are sometimes known as "Aegis programs," mainly because they use calls that have historically been documented in the Aegis reference manuals.

The availability of two complete and separate UNIX environments with Domain/OS provides maximum portability of software from Domain/OS to other BSD or SysV UNIX systems. Two key mechanisms support portability of applications programs you have created or used within Domain/OS:

- An object file *runtype* stamp that specifies the call semantics required for the program.
- An object file *systype* stamp that specifies other aspects of the program's run-time environment.

Some of the UNIX system calls use the *runtype* value to determine which environment's semantics to use for a system call that has the same name in both environments. The default value for the *runtype* of a program is the value of SYSTYPE in the program's process. You can explicitly set the stamp's value using the **-runtype** option when you compile or bind the program.

If you compile or link with the **-systype** option or if you use the **#systype** directive, the compilers and linkers create a record in the object file identifying the *systype* associated with a program. At runtime, the loader uses the *systype* stamp to set the value of the SYSTYPE environment variable for the program's process. If the *systype* isn't specified in the object file, or if the *systype* is **any**, the loader sets SYSTYPE to the value it had in the parent process.

The value of SYSTYPE determines some aspects of the program's run-time environment. For example, SYSTYPE determines whether the pathnames **/bin**, and **/usr/bin** refer to the SysV directory tree or to the BSD directory tree, as we described in Chapter 1.

A program's *runtype* stamp and *systype* stamp need not have the same value. Moreover, one or both of the associations can be dynamic; that is, they can have the value **any**, which means "the current SYSTYPE value." Consider, for example, the SysV **nm** utility, which prints the name list (symbol table) of each object file in the argument list. The **nm** object code contains a SysV *runtype* stamp, that is, it requires SysV semantics for the system routines it calls. Its *systype*, however, is **any**. Invoked from a shell where the SYSTYPE environment variable is **sys5.3**, therefore, the command

```
% nm /bin/ls
```

would print the symbol table for **/sys5.3/bin/ls**. Invoked from a shell where the SYSTYPE environment variable is **bsd4.3**, the same command would look at the **/bsd4.3/bin/ls** sym-

bol table. In both cases, however, the *runtype* is still **sys5.3**, so the **nm** utility always uses the SysV versions of the system services it calls.

2.2.3 The Compilers

Domain/OS provides compilers for programs written in the following languages:

- Domain/C
- Domain FORTRAN
- Domain Pascal
- Domain/Ada
- Domain/CommonLISP
- Domain/C++

(Third party vendors may supply other compilers; we will limit our discussion to those compilers available directly from Apollo.)

Domain compilers support industry standards for these languages. As a result, people familiar wiih these languages from other systems confront as few unknowns as possible. Also, support of industry standards facilitates porting existing source code to Apollo workstations.

The following sections briefly discuss the compilers in terms of their user interfaces. Where there is more than one interface to a compiler, we present information about the differences between them. For more details on the Apollo implementation of a particular language, refer to the documentation mentioned below.

2.2.3.1 C

The C programming language was designed to give programmers a convenient means of accessing a machine's instruction set. It is high-level enough to make programs readable and portable, but simple enough to map easily onto the underlying architecture of a machine. Its flexibility and power have made it a popular language for both systems and application programming.

We have two interfaces to the Domain/C compiler: one that is adapted for the Aegis environment, and one that is adapted for the UNIX environments. The differences between the two interfaces are largely differences in command line syntax and default behavior. These differences let you use the interface most familiar to you and make porting existing code from other machines easier. Compiler function isn't restricted by the calling environment, however; all the options available when calling the compiler with either interface are available (with syntactic differences) when calling the compiler with the other interface.

The tools (if any) that are invoked along with the compiler differ for the two compiler interfaces. For example, the UNIX interface to the C compiler (**/bin/cc**) automatically invokes the UNIX **ld** linker after executing the compilers themselves, whereas the Aegis interface (**/com/cc**) invokes the compiler without linking its output.

For more information about invoking the Domain/C compiler and the differences between the two interfaces, see the *Domain/C Language Reference* (002093).

2.2.3.2 C++

The C++ programming language was designed to simplify the process of good programming. It is a superset of C. C++ differs from C by being more expressive but requiring greater attention to object type. In effect, C++ attempts to provide programmers with a language closer to that in which they express problems and their solutions.

Like Domain C and Domain FORTRAN, Domain/C++ has two interfaces: one for the UNIX environments and one for the Aegis environment. The differences between the two interfaces are few.

For details concerning Domain/C++, please see the *The Domain/C++ Programming Language* (012780).

2.2.3.3 FORTRAN

FORTRAN was originally employed for scientific computation. It is still widely used for this purpose, but it is frequently used for other applications as well. It is one of the older, more durable high-level programming languages. FORTRAN 77, the standard upon which Domain FORTRAN is based, offers programmers the ability to write fairly structured programs using a language that resembles English.

Like the Domain/C compiler, the Domain FORTRAN compiler has two interfaces: one that is adapted for the Aegis environment, and one that is adapted for the UNIX environments. As with the C compiler interfaces, the two FORTRAN interfaces differ in ways that provide you with a familiar interface, whatever operating environment you use. In addition, the UNIX interface has options that give you all the same capabilities available through the Aegis interface.

The two FORTRAN interfaces have the same type of behavioral differences as do the two C compiler interfaces. Like the syntactic differences between the interfaces, the behavioral differences exist to give you the type of behavior that past experience with Aegis or UNIX operating systems and FORTRAN leads you to expect.

For more information on Domain FORTRAN and the two interfaces, refer to the *Domain FORTRAN Language Reference* (000530).

Ratfor, a preprocessor designed to make FORTRAN a more structured programming language, is available through the UNIX environments' interface. See Section 2.10.6.

2.2.3.4 Pascal

The Pascal programming language was originally written to be a teaching language that would encourage and facilitate the use of structured programming. In addition, it was intended to be a reliable, efficient high-level language. Since its development, Pascal has become one of the most popular structured, high-level languages.

Unlike the Domain/C and Domain FORTRAN compilers, the Domain Pascal compiler has only one interface: one accessible through the Aegis environment. For information on the Domain Pascal language and the Domain Pascal compiler, see the *Domain Pascal Language Reference* (000792).

2.2.3.5 CommonLISP

CommonLISP is a functional, or applicative, language. It has two salient features: a list-based representation of data and an evaluator, or interpreter, that treats some lists as programs. Programs and data have the same form in LISP, and thus LISP programs can easily process other LISP programs. Programs are sequences of expressions composed of function calls.

Domain/CommonLISP has one interface in Domain/OS. This interface is accessible through both UNIX environments. For more information on Domain/CommonLISP and its interface, see the *Domain/CommonLISP User's Guide* (008791), the *Domain/CommonLISP Advanced User's Guide* (015738), or the *Domain/CommonLISP Tool Reference* (015739).

2.2.3.6 Ada

Ada was developed for the US Department of Defense. Ada is better thought of as an environment for programming than simply as a programming language. Ada implementations are toolkits containing facilities for library management, compilation, program generation and analysis, debugging, source code formatting, and other functions.

Domain/Ada processes the full Ada language as specified by the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A (008684). The Domain/Ada development system is accessible through the Domain/OS BSD environment only. For complete details on the Ada development system, please see the *Domain/Ada Development System Reference* (008917).

2.3 Linking

The linkers, also referred to as the binders, combine one or more input object modules to form one output object file. One feature unique to the Domain linkers is the ability to generate object files with access to installed libraries. Although the linkers do not load routines from the installed libraries, they do look at a list of the global symbols defined by the installed libraries. The linkers check 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 linker concludes that the unresolved external reference will be resolved at run time, and therefore does not issue a warning. However, if an unresolved external reference cannot be resolved by a global symbol in the list, then the linker issues a warning.

There are two **linkers** in the Domain system: **bind** in the Aegis environment and **ld** in the UNIX environments. In essence, both linkers perform the same function: they combine one or more input object modules to form one output object file. However, there are some differences in the behavior of the linkers. For example, **ld** can produce smaller object files (by excluding relocation information from the object file); **bind** alone cannot do this. (You can make object files produced with **bind** smaller by running them through **ld** and using the **-s** option after you've generated them with **bind**.) As long as its associated operating environment is installed on your workstation, each linker is executable from either environment.

The UNIX environment interfaces to the C and FORTRAN compilers commonly invoke **ld** automatically. The Aegis environment interfaces to C and FORTRAN, and all other compilers, do not automatically send their output through a linker. The Ada compiler has two utilities, **a.ld** and **a.make**, which invoke the **ld** linker (along with performing other tasks).

The interface to **ld** differs slightly for each of the UNIX environments. This allows you to run the linker exactly as you've run it on other UNIX systems. Added command line options control Domain/OS enhancements to **ld**. These options take the form **-A** *optiontype* and give **ld** all the functions of **bind** as well as those typically found in UNIX linkers.

Both **bind** and **ld** can produce cross-target executable code. When you invoke one of the linkers for cross-development, you must make sure that your invocation includes the CPU type of the target processor. Otherwise, the object files will not be combined. When you use a cross-development switch when invoking a compiler, the linker will ensure that all libraries linked in are of the appropriate CPU type. (See the compiler option information in the language reference manuals for the correct cross-development option for your compiler interface.)

Chapter 7 of this book provides more information about **bind**. See the relevant UNIX documentation for information about **ld**.

2.4 Using Installed Libraries

An **installed library** is an object module loaded in global address space or in the address space of another program. This allows the program to execute code and access data stored in the installed library.

Installed libraries are an often-overlooked feature of the Domain/OS operating system. They provide an alternative to linking 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. An installed library stored in global address space allows for more efficient use of physical memory when two or more programs executing concurrently require the same library. Finally, the use of installed libraries makes it much easier to update the software on a workstation. When a new version of an installed library is available, you need only copy it onto the workstation. Under some conditions, a node may then have to be rebooted, but this is still much simpler than having to locate all bound images that use the library and rebind them to use the new version.

At run time, the loader matches your program's requests for service routines with the service routines available in the installed libraries.

Installed libraries are detailed in Chapter 4.

2.5 Using Library Files

A **librarian**, also referred to as an archiver, is a utility that creates, edits, and describes library files. A **library file** is a special file consisting of one or more object modules collected together for easy access by a linker. Using a 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 a linker can load a subset of them.

You can use a library file as input to another librarian command or as input to a link operation. You cannot use a library file as an installed library, and you cannot execute a library file.

The linkers accept both library files and object files as input. The linkers *unconditionally* load *all* the object modules stored in object files. However, the linkers load 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 **bind** option **-include**
- The object module satisfies an unresolved external reference

- If no input module defines a possible start address, **bind** looks for a possible start address in the unloaded library object modules

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

The Domain/OS operating system has two librarians, or archivers: **ar** in the UNIX environments and **lbr** in the Aegis environment. Both perform the same library management functions and can be used interchangeably. Using **lbr** does not earmark a library as an "Aegis" library; nor does using **ar** mark a library as a "UNIX" library. Both tools create exactly the same type of library. A library created with either archiver can be linked (using either linker) with code created using any of the three operating environments.

For more information on **ar** consult the *BSD Command Reference* or the *SysV Command Reference*. The **lbr** librarian is described in Chapter 8 of this book.

2.6 Source Code Control and Configuration Management

Source code control is the process of creating, storing, and accessing multiple versions of source code. Domain/OS supports two products that perform source code control:

- The Source Code Control System (SCCS), which is included in both the BSD and SysV environments
- The Domain Software Engineering Environment (DSEE), an optional product which can run in any Domain/OS operating environment. (The part of the DSEE software that provides source code control is known as the history manager.)

Configuration management is the process of generating executable code from one or more source files automatically. Domain/OS provides two products that perform configuration management:

- The **make** utility, which is included in both the BSD and SysV environments
- The Domain Software Engineering Environment (DSEE) configuration manager

The following sections discuss the available Domain/OS source code control and configuration management tools and their strengths.

2.6.1 Source Code Control

SCCS and the DSEE history manager both store different versions of source code. Each product takes a slightly different approach to version control, resulting in trade-offs in function for each. In the following subsections we present several areas of difference between the two.

2.6.1.1 Naming Versions and Branches

One area of difference between SCCS and the DSEE history manager involves the degree of user control available for naming versions and branches (or alternate evolutionary paths for a particular source module).

Both products automatically assign consecutive numbers to versions on an evolutionary path. These numbers give you access to different versions. In addition, the DSEE history manager lets you assign names to particular versions. Version names can be particularly helpful when you want to examine the specific version of source code that you used to construct a released product. If you've marked all of the constituent source code versions with the name **release_1.0**, for example, you don't have to remember that Release 1.0 was constructed using version 4 of source module **a.c**, version 20 of module **b.c**, version 13 of module **c.c**, and so on. SCCS does not allow you to name versions.

The DSEE history manager also gives you control over branch names. You assign a mnemonic name, such as **bugfix_branch** or **work_for_special_release**, to an alternate line of descent. SCCS automatically assigns numbers to branches; you use these numbers to identify a branch.

2.6.1.2 Accessing Versions from Outside

The DSEE history manager is very tightly integrated with the Domain/OS file system. This gives you access to any DSEE version from outside of the DSEE environment. You can read or compile any version without having to use any special commands. SCCS does not provide this level of external access.

2.6.1.3 Embedded Keywords

SCCS recognizes certain keywords embedded in source code. The system expands these keywords for you when you are reading a version's text. This can be very useful because it enables you to determine, for example, the creator of a particular version. The DSEE history manager does not provide this facility.

2.6.1.4 Constraints on Storage Type

SCCS can store only ASCII files. The DSEE history manager can store any type of file that can be used with Domain/OS (although space savings are greatest when your versions contain ASCII text).

2.6.1.5 Source Code Control and Configuration Management

The SCCS and **make** tools are completely independent of one another. Domain/OS supports extensions to **make** that give it access to SCCS files when building systems. However, SCCS cannot be thought of as tightly integrated with **make**.

The DSEE software treats source code control and configuration management as parts of the whole task of software development. The DSEE configuration manager is tightly integrated with the DSEE history manager, providing access to any version of a source file when building systems. The DSEE history manager can also use information about the components of a built system to name versions of elements.

2.6.2 Configuration Management

Each of the two configuration management tools available using Domain/OS, **make** and the DSEE configuration manager, recognize and cooperate with one of the two source code control mechanisms discussed above. The **make** utility has extensions to it that enable it to access the most recent versions of SCCS files, and the DSEE configuration manager is fully integrated with the DSEE history manager, providing the ability to build programs with any existing version of DSEE elements.

There are many differences between **make** and the DSEE configuration manager. Most of them are the result of the different needs that they address.

- The goal of **make** is to simplify the work of a single person working on a small to moderate-sized program.
- The goal of the DSEE configuration manager is to simplify and clarify the work of many people working on a large, complex program.

To understand the impact of each approach to configuration management, consider how each product behaves when invoked to construct a program.

When invoked, **make** looks at the date and time of the last modification to the program and compares it with the dates and times of the program's source files. If the program itself is older than one or more of the source files, **make** regenerates those parts of the program that are out of date and reconstructs the program. By default, the older version of the program is deleted. (You can work around this behavior by writing makefiles that save older versions of programs by renaming them or copying them to files with **.bak** suffixes, for example.)

When you issue a DSEE **build** command, the DSEE configuration manager does not make the assumption that you want to regenerate a program simply because input files have been modified. The configuration manager examines a user-supplied **configuration thread** to determine exactly which versions of constituent source files you want for your configuration of the program. Using this information, the configuration manager constructs the program

as well as a **bound configuration thread (BCT)**—that is, a record of the precise contents of that build.

Once it has generated the program, the DSEE configuration manager stores both the new build and its BCT in a directory dedicated to containing these objects. The new build and older builds coexist in this directory until a user-specified limit on the directory's capacity is reached; then the oldest builds are deleted.

The behavior of **make** is much simpler than that of the DSEE configuration manager. As a consequence, it is also rather limiting. The **make** approach assumes that you always want to build a program using the latest versions of the constituent source code; it cannot automatically reconstruct older builds (as you may want to do when fixing bugs in older, released software). Moreover, **make** assumes that you have no need of older builds, or, if you do, that you will rename them so they aren't deleted. If you are working with several other people on the same project, one of you could easily delete another's recent build accidentally. Also, the lack of a record of a build's contents (as provided by a DSEE BCT) makes it difficult to determine what went into any given **make**-generated build. You can work around these three **make** limitations, but it requires effort, care, and organization on the part of you and your coworkers.

The differing philosophies about configuration management make the differences between **make** and the DSEE configuration manager far greater than those between SCCS and the DSEE history manager—and too numerous to present here. For complete details on each utility, consult the documentation on each product listed below.

2.6.3 Documentation on Source Code Control and Configuration Management

Complete details on SCCS and **make** are included in the UNIX documentation described in the preface of this book. The DSEE product, which provides task management and release management in addition to source code control and configuration management, has a document set of four books:

- *Getting Started with the Domain Software Engineering Environment* (008788)
- *Engineering in the DSEE Environment* (008790)
- *Domain Software Engineering Environment (DSEE) Command Reference* (003016)
- *Domain Software Engineering Environment (DSEE) Call Reference* (010264)

2.7 Executing Programs

The **loader** is the Domain/OS utility that starts execution of all programs. When you enter the name of a file to be executed, the loader creates a process for your program to run in and maps that program into the process's address space. The loader sometimes also in-

stalls object files that will be referenced during program execution: the installed libraries. Another important function of the loader is to check the run-time requirements of the program to make sure it can be executed.

As we noted before, there are no loader options or other methods for explicitly controlling the loader. You may, however, control the loader indirectly by giving it instructions through the compilers and linkers. At various places throughout this manual, there will be descriptions of compiler or linker options that, in effect, allow you to control the loader. One example of your control of the loader is the **-inlib** compiler option. This compiler option directs the compiler to tell the loader to install a certain run-time library when the program is loaded.

The loader is invoked automatically every time you execute a program.

2.8 Debugging

There are two debuggers available in Domain/OS: **dbx**, which is available in the UNIX environments; and Domain/DDE, which is available in all three environments.

The Domain/OS **dbx** debugger provides all the same debugging capabilities traditionally found in UNIX systems. We have enhanced **dbx**'s usefulness by adding the display of source code to its user interface. However **dbx** cannot debug programs that take advantage of some of the unique aspects of the Domain/OS environment; for example, **dbx** does not support multiprocess or remote debugging.

The Domain/DDE debugger is a powerful debugger with many sophisticated features, including:

- A context-sensitive, menu-driven interface
- Graphic displays for source and assembly code, watched variables, and traceback
- Language-sensitive expression evaluation
- Support for multiprocess and remote debugging

If you're familiar with **dbx** but want to explore Domain/DDE's capabilities as a debugging tool, you'll find that Domain/DDE provides compatibility commands that simulate many **dbx** commands. These commands give you the same functions as **dbx** and thus ease transition to Domain/DDE. (However, the command output differs, since the two debuggers have differing displays.)

The **dbx** debugger is described in the UNIX documentation described in the preface of this manual. Domain/DDE is fully described in the *Domain Distributed Debugging Environment Reference* (011024).

2.9 Analyzing Program Performance

There are several performance analysis tools available in the Domain/OS programming environment:

- **prof**, which gives a flat profile of C and FORTRAN programs' execution (available, in slightly differing configurations, as a standard part of both the BSD and SysV environments).
- **gprof**, which displays call graph profile data (available as a standard part of both the BSD and SysV environments).
- **hpc**, which produces a histogram of the program counter during program execution (available as a standard part of all three environments).
- **Domain/PAK**, a set of tools that provides detailed analysis of program performance, going far beyond the information available through **prof** and **gprof**. (Although **hpc** is a standard part of all three programming environments, it is also a component of Domain/PAK.) Domain/PAK is an optional program analysis kit that can run in all three environments.

2.9.1 prof and gprof

The Domain/OS utilities **prof** and **gprof** behave exactly as they do in other UNIX environments. If you are familiar with these routines from using other systems, you will find it easy to use them in the Domain/OS environment. The **prof** profiler is documented in the *BSD Command Reference* and the *SysV Command Reference*. The **gprof** profiler is documented in the *BSD Command Reference*.

One drawback of **prof** and **gprof** is that they are not designed to work in an environment with global libraries. They cannot profile calls to routines in global libraries, such as those used in the Aegis programming environment. Domain/PAK profiles calls to routines in global libraries.

Programs analyzed with **prof** or **gprof** must be compiled specifically with profiling in mind. The compiler interfaces recognize options that generate code that **prof** and **gprof** can profile.

2.9.2 The Domain Performance Analysis Kit (Domain/PAK)

Domain/PAK consists of three tools that help you analyze the performance of your applications. Each of the tools examines performance in a different degree of detail. You can use one or several of these tools to examine an application. Programs that use the Domain/PAK tools do not have to be specially compiled.

Following is a brief description of each of the tools in Domain/PAK. For full information about Domain/PAK see *Analyzing Program Performance with Domain/PAK* (008906).

2.9.2.1 dspst

The **dspst** program continuously monitors the CPU and I/O usage of all the processes running on a node. It displays the relative percentage of CPU used by each process, and the total system paging, disk, and network I/O.

2.9.2.2 dpat

The **dpat** program periodically samples a program's call/return stack while the program is executing. These samples allow **dpat** to estimate the relative amount of time each procedure is **active**, that is, the relative amount of time spent either in the procedure itself or in the procedures it calls directly or indirectly. Therefore, **dpat** can identify procedures that consume a relatively large amount of time, *irrespective* of whether those procedures consume time directly, or instead, tend to invoke other time-consuming procedures or system services. For example, **dpat** can highlight a procedure that performs a great deal of I/O, even though the program counter is rarely within that procedure itself, because the I/O services it invokes cause the procedure to be *active* for a relatively large amount of time.

Call/return stack samples also allow **dpat** to break down the time a procedure is active according to **context**, that is, according to which procedure called it directly or indirectly. Thus, **dpat** can distinguish between the performance of procedure X when called by Y, and the performance of X when called by Z.

dpat includes a variety of functions to analyze the data collected during the sample phase.

2.9.2.3 hpc

The **hpc** program monitors a limited range of a program at a very high rate. It produces a histogram of the program counter during program execution, thus helping you locate the most time-consuming portions of your program.

The **hpc** utility samples the program counter at the user specified sampling rate. It groups the samples into "buckets." The larger the program, the more source lines grouped per bucket. The smaller the range measured, the finer the granularity of an **hpc** report.

By default, **hpc** does not measure time spent performing system calls (file I/O, graphics, and so forth). Therefore, if your program makes heavy use of system services, **dpat** will be more helpful than **hpc**.

2.9.2.4 Using the Domain/PAK Tools Together

To use the Domain/PAK tools, you might first use **dspst** to get an overall idea of how an application is using system resources. Then you would use **dpat** to see which particular parts of your program are consuming resources. If the problem is in compute-bound procedures, then you could use **hpc** to look at the particular statements using CPU time. If the problem is related to I/O, you could use **dpat**'s interactive analysis features to see how your program might be modified to reduce I/O.

2.10 Using Other Programming Tools

Many of the standard tools of the Domain/OS programming environment can help you develop programs. Here we present several of these utilities whose value to programmers is especially noteworthy.

2.10.1 The lex Lexical Analyzer Generator

The **lex** utility is a lexical analyzer generator which runs in both the UNIX environments. Our implementation of them is identical to that in other UNIX systems.

The **lex** utility is a program generator designed for lexical processing of character input streams. You supply a high-level, problem-oriented specification for character string matching, and **lex** produces a C program that recognizes regular expressions.

For more information about **lex**, see the UNIX documentation described in the preface of this manual.

2.10.2 The yacc Parser Generator

The **yacc** parser generator runs under both UNIX environments. Like **lex**, **yacc** is implemented just as it is in other UNIX systems.

The **yacc** utility is a general tool for parsing the input to a computer program. You give **yacc** a specification of the input process that includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. The **yacc** output is a parser that controls the input process.

The **lex** and **yacc** utilities are designed to be used together. You use **lex** as a preprocessor for **yacc**. In effect, **lex** handles the implementation of your program's lexical rules, and **yacc** handles the implementation of your program's grammatical rules.

The **yacc** utility is fully described in the UNIX documentation described in the preface of this manual.

2.10.3 The M4 Macro Processor

The M4 macro processor enables you to tailor a programming language for a particular application or to make it more readable. In addition to replacing one text string (or macro) with another text string, M4 can pass arguments to macros, conditionally expand macros, perform arithmetic, and manipulate files.

M4 is particularly well suited as a front end for Ratfor, C, and other functional languages. M4 is part of both UNIX environments.

For more information on M4, please see the UNIX documentation described in the preface of this manual.

2.10.4 The xar Tool and Compound Executables

When you are writing programs to run on all Apollo workstations, you may find it inconvenient to have one separate file containing the executable code for Series 10000 workstations and another separate file containing the executable code for all other types of workstations. In such situations, you can use the **xar** command. This utility, which is part of all three operating environments, combines the two executables in one file (known as a compound executable).

The Domain/OS loader recognizes compound executables created with **xar**. Whenever you invoke a compund executable, the loader determines which object module of the file it should load in memory, based on whether the code is to execute on a Series 10000 or 680x0-based workstation. No special instructions to the loader are required.

For more information on **xar** and compound executables, the *Series 10000 Programmer's Handbook* (011404).

2.10.5 The lint C Program Checker

The **lint** program examines C language source programs, detecting bugs and obscurities. It enforces the type rules of the C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. The **lint** program can also detect some wasteful or error-prone (but legal) constuctions. It accepts multiple input files and library specifications and checks them for consistency.

The Domain/OS programming environment includes two versions of **lint**: one historically shipped with BSD operating systems, and one historically shipped with SysV operating systems. Both versions of **lint** are documented in the *Domain C Language Reference* (002093).

2.10.6 The Ratfor FORTRAN Preprocessor

The Ratfor language makes up for some of the deficiencies of the FORTRAN language. Among other things, it provides

- statement grouping
- **if-else** and **switch** for decision-making
- **while**, **for**, **do**, and **repeat-until** for looping
- **break** and **next** for controlling loop exits
- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- symbols such as >, >=, and < instead of .GT., .GE., and .LT.
- **return** (expression) statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

The Ratfor compiler (actually, a preprocessor which translates programs written in Ratfor into FORTRAN) is available in the BSD operating environment. Ratfor is documented in the *Domain FORTRAN Language Reference* (000530).

Chapter 3

Object Files in Domain/OS

Object files are the files created by Domain compilers, assemblers, and linkers (**bind** and **ld**). All of the following are object files:

- Executable programs
- Installed libraries
- The input files that a linker combines to produce another object file
- The input files that a librarian combines to produce a library file

You can use an object file as an executable program if it has no unresolved references. Furthermore, options to **bind** and **ld** allow you to use their output object files as executable programs even if the files contain unresolved references.

Chapter 4 describes how to use object files as installed libraries.

You can use any object file created by a Domain compiler, assembler, or **bind** as input to a linker. You can use an output file created by **ld** with the **-r** option as input to a linker. Chapter 7 describes the use of **bind**. See the UNIX documentation described in the preface of this manual for a description of **ld**.

Chapter 8 describes how to use object files as input to a librarian.

This chapter is an overview of object files and how they're manipulated in Domain/OS. It provides a background useful for understanding the behavior of Domain compilers, linkers, the Domain loader, librarians, installed libraries, and any tools that manipulate object files.

3.1 The Object File Format

At SR10, all Domain object files are in the common object file format (COFF). COFF is a standard format instituted by AT&T. Most vendors of UNIX System V systems, including Apollo, use COFF as their object file format. Chapter 6 describes the COFF format in detail.

Prior to SR10, Domain compilers and linkers produced object files in the OBJ format. OBJ is incompatible with COFF, but we provide the **obj2coff** utility, documented in help files and man pages, for converting OBJ object files to COFF object files.

Every COFF file is composed of several parts. The AT&T standard COFF template specifies parts that must appear in every COFF file, describes the structure of optional parts that may appear, and provides a structure for adding additional, vendor-specific parts.

3.1.1 The AT&T COFF Template

The AT&T COFF template specifies parts that each COFF file must have. Every COFF file must have the following parts, in order:

- A file header
- A table of section headers
- At least one section

The COFF template specifies the formats for each of these required parts. We describe them in Chapter 6.

In addition, a COFF file may have any of the following parts, in order:

- An optional header, between the file header and the table of section headers
- Additional sections, following the table of section headers
- Relocation tables
- Line number tables
- A symbol table
- A string table

Ordinarily, compiler options, linker options, and commands such as **strip** determine whether these optional parts are included.

The COFF template defines three types of sections (text sections, initialized data sections, and uninitialized data sections), and specifies the formats for relocation tables, line number tables, symbol tables, and string tables. We describe the section types and the table formats in Chapter 6.

The COFF template places no restrictions on the format or content of the optional header. The template also allows a compiler and/or linker to define additional sections and to include them in COFF files.

Figure 3-1 illustrates the parts of a COFF file, in order.

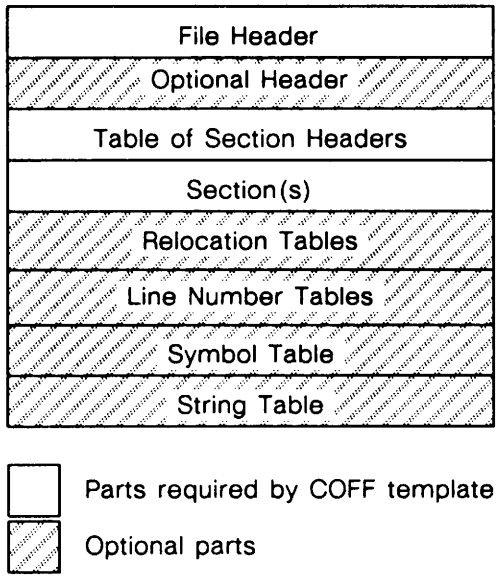


Figure 3-1. Parts of a COFF File.

3.1.2 The Apollo Implementation of COFF

Since SR10, the object files produced by all Domain compilers, linkers, and assemblers comply with the AT&T COFF template. As required by the template, all Apollo COFF files have the following parts, in order:

- A file header
- A table of section headers
- At least one text section

In addition, all Apollo COFF files have the following parts:

- An Apollo specific optional header, between the file header and the table of section headers
- A section named **.unwind**

We define the formats for the Apollo header and for the **.unwind** section in Chapter 6.

3.1.3 Overview of the Parts of a COFF File

Sections are the heart of the COFF file. A section may be one of several types. The COFF template defines text sections, initialized data sections, and uninitialized data sections; a compiler or linker may define additional types of sections.

A text section is a read-only section. Typically, a program's machine instructions are stored in text sections. Data sections are read-write. Typically, a program's static data are stored in initialized data sections and its uninitialized global data are stored in uninitialized data sections. By default, compilers and assemblers create a text section named **.text** to store a program's machine instructions, a data section named **.data** to store a program's static data, and an uninitialized data section named **.bss** to store a program's uninitialized global data.

Every COFF file has at least one section, to store the program's machine code and its data. The number of sections allowed in a COFF file is virtually unlimited, and the COFF file template requires no specific structure for additional section types. The compiler, therefore, is free to create several sections to store the program, and to create one or more additional sections to store any other information about the file. As we describe in Chapter 6, all Domain compilers take advantage of this freedom and may create several additional sections to store information about the file.

The other parts of the COFF file contain information to support the sections.

- The **file header** and the Apollo specific **optional header** are records containing some information about the file as a whole.
- A **section header** specifies the size, location, and type of a section; each section in the file must be described by a section header.
- **Relocation tables** identify address references that the linker and/or loader must change as they reposition sections.
- The **line number table** maps source code line numbers to the addresses of their corresponding instructions.
- The **symbol table** stores information about program symbols, such as their data types and values.

- The **string table** stores the character strings to which other parts of the COFF file refer.

3.2 Object Files and Virtual Address Space

The compilers, the linkers, and the loader all assign **virtual addresses** to locations within the object file. They store these virtual addresses in the table of section headers. Virtual addresses represent the addresses at which the loader stores the program's parts.

Compilers assign virtual addresses to any sections that require them. A section will have a virtual address for either of the following reasons:

- The section will be loaded.
- Another part of the object file, aside from the section header, refers to a virtual address within the section.

For example, all text sections are loaded, so they will have virtual addresses. The **.rwdi** section (described in Chapter 6) is not loaded, but the relocation tables refer to virtual addresses within **.rwdi**. The **.rwdi** section, therefore, will have a virtual address. The **.sri** section is not loaded, and no other part of the object file refers to a virtual address within **.sri**, so it will not have a virtual address.

Figure 3–2 shows how the compiler maps its output object file into virtual address space.

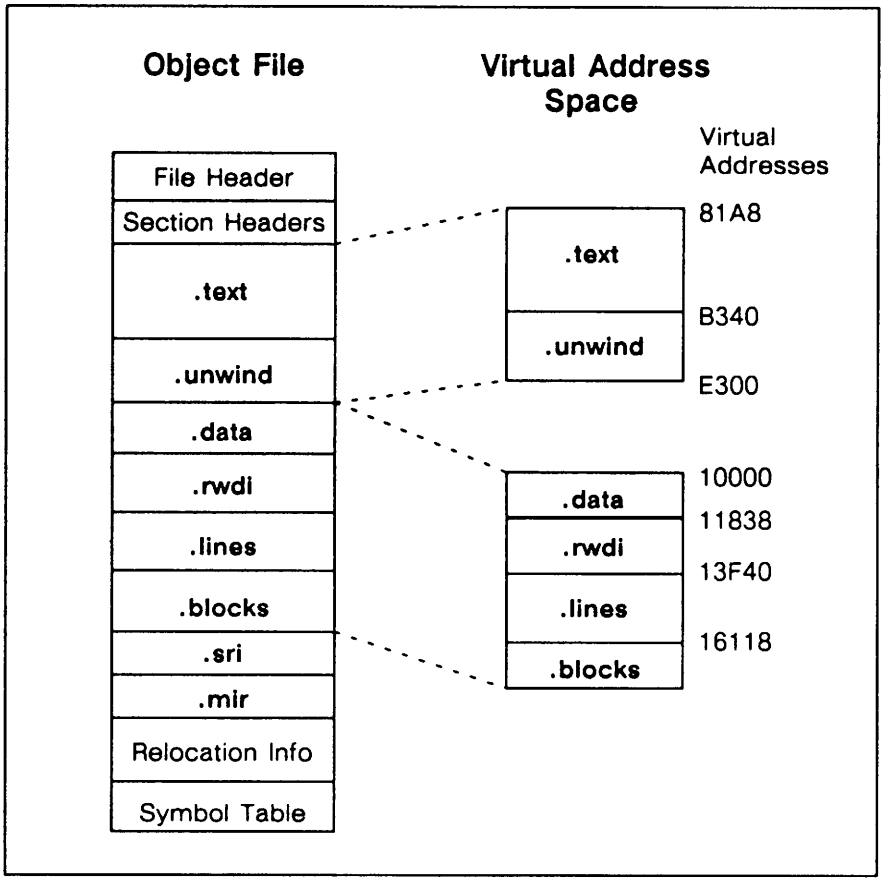


Figure 3-2. An Object File and its Mapping into Virtual Address Space

The compilers and linkers assign the virtual addresses that would probably be correct if the program were loaded immediately after compiling or linking. These virtual addresses change when the object file is linked with other object files and, perhaps, when the object file is loaded. Section 3.4 describes how linking changes the virtual addresses of the two component object files.

All references to virtual addresses are called **relocatable references**. The term **relocatable** refers to the variable nature of virtual addresses, that is, the fact that they may change during linking or loading.

Program instructions use virtual addresses to point to other locations in the program. Object file parts such as the section headers also use virtual addresses to point into sections.

If the file uses **position-independent code**, the loader may assign new virtual addresses to the sections that it loads. If the file uses **absolute code**, sections are loaded at the virtual addresses that were assigned before loading. Section 3.3 describes position-independent and absolute code.

3.3 Absolute and Position-Independent Code

Domain compilers generate two types of code: absolute code (**ac**) and position-independent code (**pic**). Compiler options, documented in the Domain language reference manuals, determine whether code is absolute or position-independent.

3.3.1 Absolute Code

Absolute code was designed for programs whose virtual addresses won't change at load time: the loader installs sections at the virtual addresses assigned by the last compile or link.

Since an absolute code program's relocatable references won't change at load time, the compiler can store those references in text sections (which are read-only), with the program's machine code. This usually produces the most efficient code.

With an absolute code file, you can always use linker options or the **strip** command to remove the relocation information and symbol table from the file before load time. This allows you to reduce the size of your object files.

Domain/OS allows your programs to reference machine instructions and data that are stored in shared libraries. (We discuss this in more detail in Chapter 4.) You can specify those shared libraries either at compile time or at link time. If an absolute code program references any symbols that are defined in shared libraries, however, you *must* specify those installed libraries *at compile time*. Compiler options, documented in the Domain language reference manuals, describe how.

Code that will run on 680x0-based workstations is, by default, absolute.

3.3.2 Position-Independent Code

Position-independent code was designed for programs whose virtual addresses may change at load time.

Since a position-independent code program's relocatable references may change at load time, the compiler must store those references in data sections, which are read-write. At load time, the loader writes the correct address references. At run time, machine instructions that refer to virtual addresses go through a level of indirection to find those addresses in a data section. (It's often more efficient, in position-independent code, to refer to another program location as an offset from the current program location, rather than as a virtual address. If the two program locations are in the same section, the offset between them will not change as the loader repositions sections.)

In order to change a program's virtual addresses, the loader requires that the program meet all of the following conditions:

- Contains position-independent code
- Retains its relocation information
- Retains its symbol table

For example, the loader must be free to change the virtual addresses in any object files that will be dynamically loaded, such as dynamically loaded libraries (described in Chapter 4) and type managers. Such a program must contain position-independent code and must retain its relocation information and symbol table.

NOTE: By default, **ld** removes relocation information. The **-r** option instructs **ld** to retain the relocation information.

In order to run on Series 10000 workstations, a program must contain position-independent code. Its virtual addresses need not change at load time, however, so it need not retain its relocation information and symbol table. All Domain compilers create position-independent code when compiling for Series 10000 workstations. Compiler options, documented in the various compiler reference manuals, allow you to create position-independent code for 680x0-based workstations.

3.4 Linking Object Files

The Domain linkers combine two or more component object files to create a new object file.

A flag in the section header (described in Chapter 6) tells the linkers how to combine a section with other sections of the same name. Ordinarily, the linker concatenates identically named sections. The **STYP_OVERLAY** flag in the section header, however, instructs the linker to overlay sections. If, for example, the **STYP_OVERLAY** flag is set in the section header for a section named **test**, the linker will assign the same virtual address to each section named **test**. No more than one of the **test** sections may contain initialized data; the rest must be uninitialized.

Consider, for example, linking three files. Each of the three files has a **.text** section. In each file's **.text** section header, the **STYP_OVERLAY** flag is not set. One of the three files has a **.data** section. Each of the three files has an **xyz** section. In each file's **xyz** section header, the **STYP_OVERLAY** flag is set.

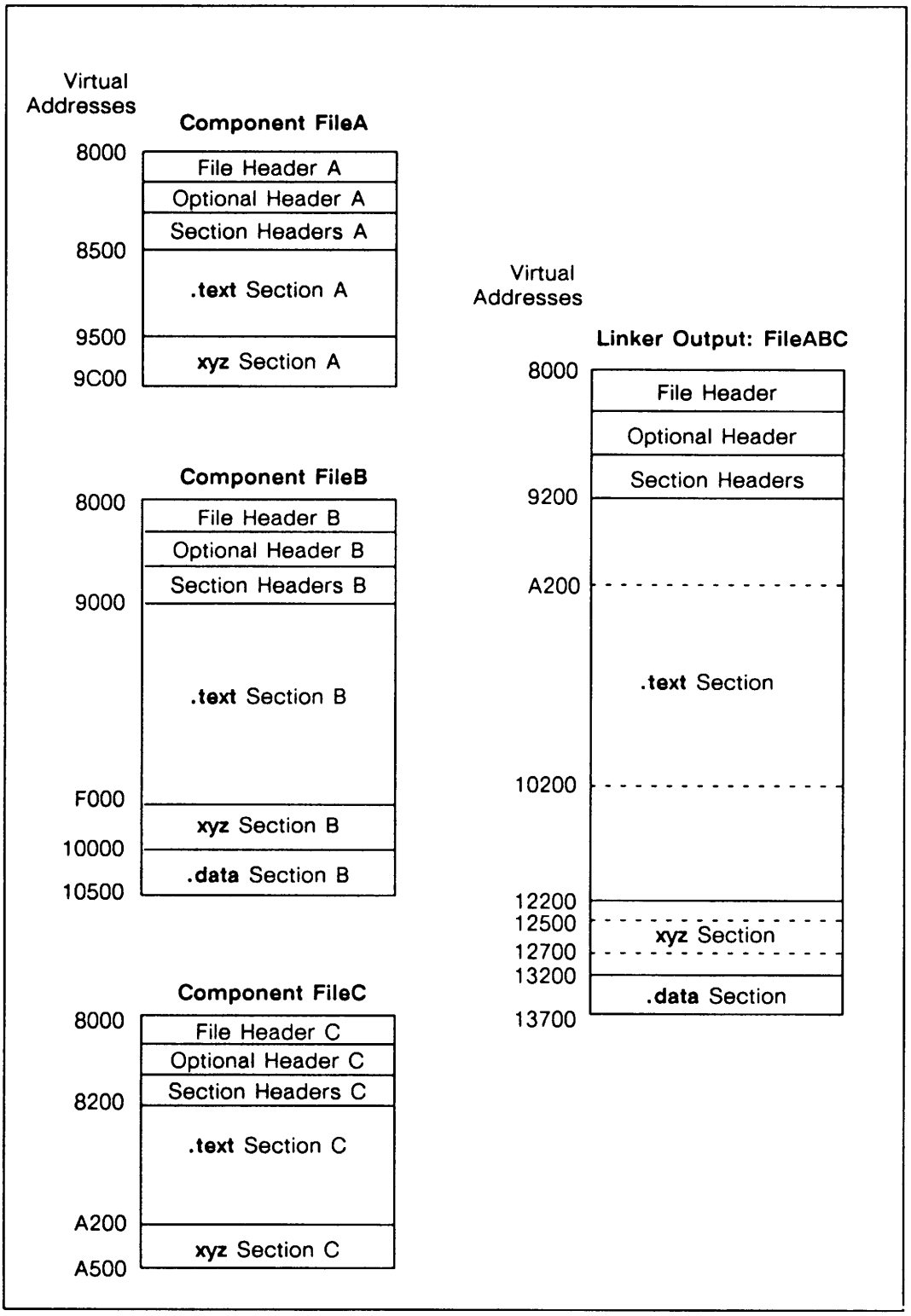


Figure 3–3. A Linker Example

As Figure 3–3 shows, the object file produced by the linker has one **.text** section, one **.data** section, and one **xyz** section. The **.text** section is a concatenation of the component **.text** sections. The **.data** section is identical to the single **.data** section in the component file. The **xyz** section is an overlay of the three component **xyz** sections. (If you read the virtual addresses to determine the size of each section, you can see that FileABC's **.text** section is as large as all three component **.text** sections combined, but that FileABC's **xyz** section is only as large as the largest of the three component **xyz** sections.)

As you can see in Figure 3–3, the linker creates new virtual addresses for its output file. The linker must therefore change all relocatable references within the program, using the following algorithm:

$$VA_{final} = VA_{init} - (SVA_{init} - SVA_{final})$$

where VA_{final} is the virtual address, after linking, of the referenced item

VA_{init} is the virtual address, before linking, of the referenced item

SVA_{init} is the virtual address, before linking, of the beginning of the section containing the referenced item

SVA_{final} is the virtual address, after linking, of the beginning of the section containing the referenced item

For example, consider linking the three files illustrated in Figure 3–3. The program contains a reference to symbol **sample_var** at address 10254. Address 10254 is within FileB's **.data** section. To determine the virtual address that **sample_var** will have after linking, the linker performs the following calculation:

$$VA_{final} = 10254 - (10000 - 13200)$$

where VA_{final} is the virtual address of **sample_var**, after linking

10254 is the virtual address of **sample_var**, before linking

10000 is the virtual address of FileB's **.data** section, before linking

13200 is the virtual address of FileB's contribution to the **.data** section, after linking

Relocation information, described in Chapter 6, identifies the relocatable references.

The linker writes a new file header and section header table reflecting the structure and content of the sections in the new object file.

For more information on **bind**, the Domain linker, see Chapter 7. For more information about **ld**, the UNIX linker, see the BSD or SysV documentation described in the preface of this manual.

3.5 Loading Object Files

The loader installs only the text and data sections of an object file, as identified by a flag in the section header. The **STYP_TEXT** flag identifies a text section; the **STYP_DATA** flag identifies a data section.

Figure 3–4 shows an object file and its installed image. Note that the **.data** section is positioned on a segment boundary, leaving unused space between it and the end of the **.unwind** section. The image itself also begins on a segment boundary.

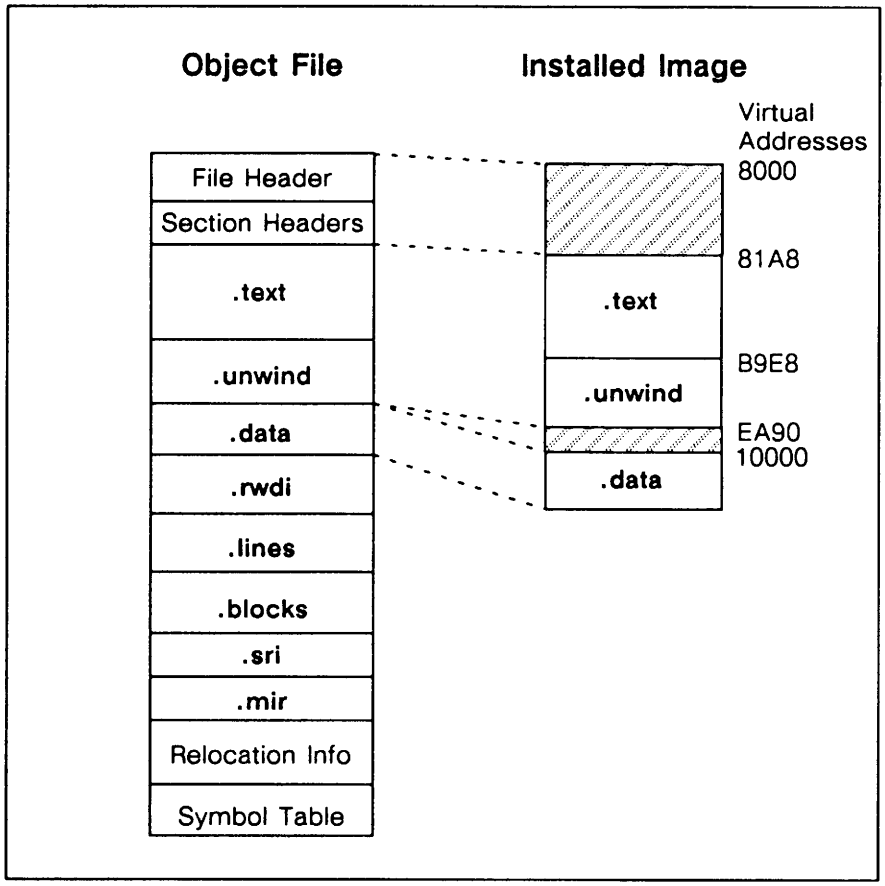


Figure 3–4. An Object File and its Installed Image

If a program uses absolute code, the loader loads each text and data section at the virtual address assigned in the object file.

If a program uses position–independent code, the loader may reposition sections. The loader keeps track of the virtual address of each section both before and after loading. As

it copies or expands data into data sections, the loader uses the same algorithm that the linkers use to change all relocatable references:

$$VA_{final} = VA_{init} - (SVA_{init} - SVA_{final})$$

where
- VA_{final} is the virtual address, after loading, of the referenced item
- VA_{init} is the virtual address, before loading, of the referenced item
- SVA_{init} is the virtual address, before loading, of the beginning of the section containing the referenced item
- SVA_{final} is the virtual address, after loading, of the beginning of the section containing the referenced item

Relocation information, described in Chapter 6, identifies the relocatable references.

The loader installs an object file in stages. First, the loader maps all the text sections into memory and allocates space for all initialized and uninitialized data sections. Next, it writes zeros into the space allocated for all data sections marked with the **STYP_BSS** or **STYP_ZERO** (described in Chapter 6) section header flags. It copies the contents of all uncompressed, initialized data sections into the space allocated for those sections. Finally, it uses the information in the **.rwdi** section (also described in Chapter 6) to write all the data stored in compressed form.

Chapter 4

Installed Libraries

You can enable a running program to access another object file's routines and data. An object file that is available to one or more running programs is known as an **installed library**.

You can use any object file as an installed library, if the object file contains position-independent code and if its global symbols are marked as accessible to other object files. Section 4.1 describes these requirements and how to meet them.

Declaring an object file as an installed library makes it available for running programs to use. You can declare an installed library available to only one process or you can declare it available globally to all processes. The library's type, described in Section 4.2, determines which processes have access to it. Known Global Tables, described in Section 4.3, provide that access. Sections 4.4 and 4.5 describe how to declare an object file as an installed library.

> **NOTE:** Don't confuse installed libraries with the library files created by **ar** and **lbr**. Installed libraries, the topic of this chapter, are ordinary object files. Library files are collections of object modules in a special format.

4.1 Creating Files for Use as Installed Libraries

We ship Domain/OS with several files created specifically for use as installed libraries. You can use any object file as an installed library, however, as long as it meets the following conditions:

- **It contains position-independent code.** All object files that will run on Series 10000 workstations meet this condition. To produce position-independent code in

object files that will run on other workstations, use compiler options documented in the Domain language reference manuals. Chapter 3 of this manual describes position-independent code.

- **Its global symbols are marked as accessible to other object files.** If the file's global symbols are marked as accessible, a program can reference any of the functions in the library and any of the data that are represented in the COFF symbol table. By default, an object file created with a Domain compiler or with **ld** meets this condition. If you are creating an object file with **bind**, you can mark its global symbols as accessible by using options described in Chapter 7.

  You may want to mark one or more of the object file's sections as available, as well. A program can refer to entire sections in the installed library only if those sections are marked as available. For example, data stored in a FORTRAN COMMON block appear in the object file in a single section. If your installed library includes a FORTRAN COMMON block, your programs can reference the entire block only if you mark the block's section as available. Options to the linker instruct it to mark sections as available. Both **bind** and **ld** can mark sections this way.

4.2 Types of Installed Libraries

You can declare an object file as one of the following general types of installed libraries:

- **Shared library**—available only to the process(es) that declare it explicitly.
- **Globally known library**—declared once and available to all processes.

Section 4.4.2 describes how to declare an object file as a shared library. Section 4.5.1 describes how to declare an object file as a globally known library.

When and where a library is installed depends on its type. A shared library is installed at load time in the address space of the procedure that declares it.

Section 4.4 describes shared libraries in more detail.

A globally known library may be one of the following three types:

- **Load-time library**—installed at load time in each process that accesses it.
- **Dynamic library**—installed during run time in each process that accesses it.
- **Global library**—installed at boot time in global address space.

Section 4.5 describes the types of globally known libraries in detail.

4.3 Known Global Tables

Domain/OS maintains a **known global table (KGT)** of symbols to which all processes have access. The KGT lists the names and addresses of all symbols defined in globally known libraries. The loader makes calls to the KGT when it installs a program, in order to find external symbols.

For each process that uses shared libraries, Domain/OS maintains a **process known global table (process KGT)**. (For efficiency, the system does not create a KGT for every process.) The process KGT lists the names and addresses of the symbols that are available only to object files in the process, that is, all symbols defined in shared libraries declared by the process and all symbols defined in the main program. At run time, the loader searches the process KGT for each external symbol before searching the KGT. If a symbol appears in both tables, therefore, the loader uses the listing in the process KGT.

Figure 4–1 shows the loader's search path.

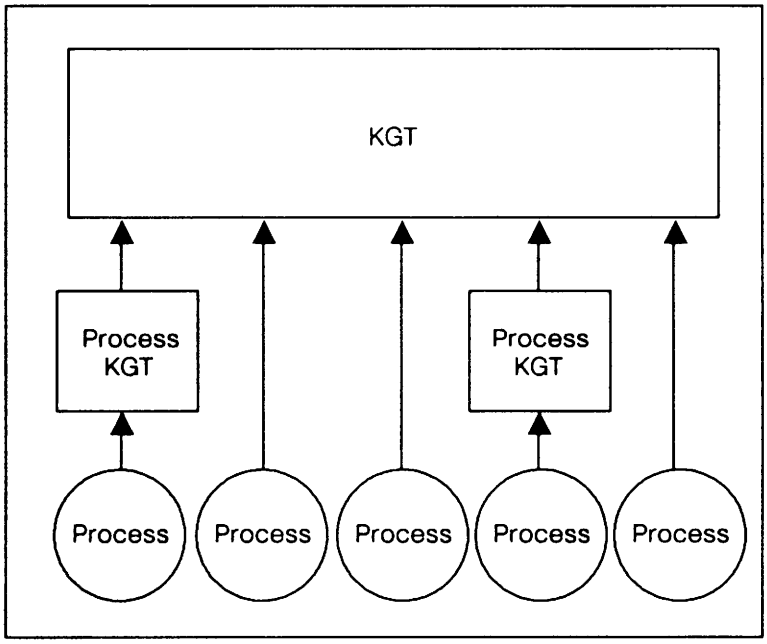


Figure 4–1. The Loader's Search Path for External Symbols

If the KGT lists a symbol more than once, the loader uses the listing of the symbol most recently installed before run time.

4.4 Shared Libraries

A shared library is an object file that is available only in the process or processes that declare it. Any programs running in those processes can access variables and routines in the shared library. Programs in other processes can't access the library.

In the address space of each process, the loader installs a copy of each shared library declared by that process, as follows: When a process starts, the loader installs copies of any load-time shared libraries. Next, the loader installs the main program that will run in the process, and begins executing the program. If two processes declare the same shared library, the loader installs a separate copy of the library in each of the two processes. When the main program first accesses a dynamic library, the loader installs a copy of that library.

As we describe in Section 4.6, load order determines which of an object file's marked symbols and sections are available to which other object files. If you want to override the default load order (load-time shared libraries, then main program, then dynamic shared libraries), you can use the **loader_$load** call to install an object file rather than relying on the loader. The **loader_$load** call is documented in the *Domain/OS Call Reference, Volume 1* (007196).

4.4.1 Knowledge of Shared Libraries

If a process uses the **fork** system call to create a child process, the parent's shared libraries are known to the child process, as well. If the parent process uses **pgm_$invoke** to create a child process, however, the parent's shared libraries are *not* known to the child process.

4.4.2 Declaring a Load-Time Shared Library

The **-inlib** option instructs the loader to use an object file as a shared library. You can use the **-inlib** compiler option when you compile the program that accesses the library, or you can use the **-inlib** option to **bind** or the **-A inlib** option to **ld** when you link the program.

In the following example, the program **main_program.bin** declares **shared_lib.bin** as a shared library, at compile time:

```
$ /com/cc main_program -inlib shared_lib.bin
```

In the following example, the program **large_program.bin** declares **shared_lib.bin** as a shared library, at link time:

```
$ ld -A inlib shared_lib.bin -o large_program part1.bin part2.bin
```

If a program uses absolute code and refers to data in a shared library, you *must* declare the shared library at compile time. (Chapter 3 describes absolute code, the default for programs that run on 680x0-based workstations.)

If you do not need to use **-inlib** at compile time, you may want to save keystrokes by using it at link time: if a program is composed of several object modules that all refer to a shared library, you can declare the library once, at link time, instead of declaring it when you compile each module.

4.4.3 Declaring a Shared Library at Run Time

We provide an alternate mechanism for installing shared libraries for compatibility with pre-SR10 versions of Domain/OS. To declare an object file as a shared library at run time, issue the **inlib** shell command before invoking the program.

In the following example, the current process declares **bar.bin** as a dynamic shared library:

```
$ inlib bar.bin
```

4.5 Globally Known Libraries

As we described in Section 4.2, there are three types of globally known libraries: global libraries, dynamic libraries, and load-time libraries. All three types of globally known libraries are available to any running program. Their symbols appear in the KGT.

4.5.1 Declaring a Globally Known Library

To declare an object file as any of the three types of globally known libraries, list its name in the file **/etc/sys.conf**. Domain/OS ships with several object files already declared as globally known libraries. You can declare other globally known libraries by adding their filenames to **/etc/sys.conf**.

The following is a portion of a sample **/etc/sys.conf** file:

```
...
lib gprlib, global
lib shlib, global, not_16mb_va
lib dblib, global, not_16mb_va
lib x25lib, global, optional
lib dialoglib, global, optional, not_16mb_va
lib userlib.global, global, optional
lib userlib.private, optional
...
```

At boot time, the system reads **/etc/sys.conf** and records each globally known library's symbols in the KGT.

Modifiers to an object file's listing in **/etc/sys.conf** determine what type of globally known library it is. You can list an object file in **/etc/sys.conf** with one or more of the following modifiers, separated by commas:

dynamic	The object file is a dynamic library, described in Section 4.5.3.
global	The object file is a global library, described in Section 4.5.4.
optional	If this library does not exist, the loader will not report an error.
not_16mb_va	On machines with 16 MB of virtual address space (DN300, DSP80), this library is not global; instead, it is a load-time library.
not_64mb_va	On machines with 64 MB of virtual address space (the DN330, DSP90, DN5x0, and some DN3000 workstations), this library is not global; instead, it is a load-time library.

4.5.2 Load-Time Libraries

An object file listed in **/etc/sys.conf** with neither the **dynamic** nor the **global** modifiers is a load-time library. The loader installs a copy of the library in the process space of every program that references any of the library's external symbols. Each process that accesses the library gets its own copy of the library. Changes made to the library by one process, therefore, do not affect the library in another process. The loader installs the library when it installs the program that accesses the library.

4.5.3 Dynamic Libraries

An object file listed in **/etc/sys.conf** with the **dynamic** modifier is a dynamic library. A dynamic library behaves as a load-time library, except that the loader does not install it until the program calls it, at run time. When a running program calls a symbol defined in a dynamic library, the loader interrupts the program flow and installs the library in the program's process. If a second program accesses a symbol in the same dynamic library, the loader installs a separate copy of the library in that program's process. Changes made to the library by one process do not affect the library in another process.

4.5.4 Global Libraries

An object file listed in **/etc/sys.conf** with the **global** modifier is a global library. At boot time, the loader installs all global libraries in global address space, where they are accessible to all running programs. The loader does not install copies of global libraries in any processes: all processes that access the library access the single copy in global address

space. (As described in the following paragraphs, however, the loader may install copies of individual data sections.)

The loader imposes the following rules about how a process can access sections within a global library:

- Text sections in global libraries, like text sections in any object file, are read–only. Any process can read the information in a global library's text section, but no process can write it.
- If a data section's name is **.data** or if its name ends in **_pure_data$**, the loader initializes the data section and then marks it as read–only. Any process can read the data, but no process can write it.
- If a data section's name ends in **_global_rw$**, any process can read or write the data.
- For all other data sections, the loader initializes the section to zero, regardless of the intial values specified in the object file. The loader gives each process its own copy of the data section, which it can read or write. Even though the data section is part of a global library, the loader installs a separate copy of the data section into each running process. You can create an initialization routine that an individual process can run to initialize the data in the section.

4.6 Running Programs with Installed Libraries

Once the loader installs a library in a process, any program in the process can refer to any marked symbol or section in the library. Your programs can use variables defined in the library and call routines that appear in the library. Code within the installed library can, in turn, call routines that appear in the main program or in other installed libraries.

An installed library cannot necessarily access data defined in other object files, however. Access to data is determined by load order: each object file's marked data is available to all subsequently loaded object files. It is not available to previously loaded object files.

The loader does not necessarily load installed libraries in the order in which you specify them. The only guarantee about load order is that load–time shared libraries are installed before the object file that requires them, and that dynamic libraries are installed after the object file that requires them.

You can guarantee load order if each library is an object file that declares one shared library. In the following example, the program **my_program** uses the shared libraries **highlib** and **lowlib**. The **lowlib** library must be installed first, however. If **my_program** declared both **highlib** and **lowlib** as shared libraries, the loader would not guarantee load

order. Instead, **lowlib** declares **highlib** as a shared library, and **my_program** declares **lowlib**:

```
$ ld -o /lib/highlib c.bin d.bin -A inlib /lib/lowlib
$ ld -o my_program a.bin b.bin -A inlib /lib/highlib
```

The **-A inlib** option on the second command line means that before loading **my_program**, the loader must install **highlib**. The **-A inlib** option on the first command line means that before loading **highlib**, however, the loader must install **lowlib**.

If an installed library requires access to data in the main program, the main program must be installed before the library. To ensure that load order, you can either make the libraries dynamic or you can call **loader_$load** from the main program. If you make the libraries dynamic, the loader installs them at run time, after installing the main program. The **loader_$load** call bypasses the loader and installs the libraries explicitly, after the loader has installed the main program. The **loader_$load** call is documented in the *Domain/OS Call Reference, Volume 1* (007196).

If an installed library contains an entry point, the loader executes that code when it loads the library. If you use **loader_$load** to load the library, however, the library's entry point code is not automatically executed. To call the module, use **loader_$lookup_start_addr** to find its start address.

Chapter 5

Calling Conventions

Apollo uses standard calling conventions so that program modules may call each other, whether or not they are written in the same language. Programs that run on 680x0 processors use one set of calling conventions; programs that run on Series 10000 processors use another set of calling conventions. Domain FORTRAN, Domain Pascal, Domain/C and Domain/Ada all use these two sets of standard calling conventions.

Understanding the calling conventions allows you to write assembly language routines that can be called from higher-level language routines and helps you write applications that mix higher-level languages. Furthermore, an understanding of the conventions allows you to understand compiler generated code, and can be an aid to low-level debugging.

The calling conventions discussed in this chapter pertain only to calls to subprograms external to the calling program's compilation unit. A FORTRAN subroutine and a Domain/OS system call are necessarily external to a C main program, as is a C function whose code is contained in a separate file. With internal routines, which are called from within the compilation unit that defines them, the compiler is free to optimize the argument passing conventions.

5.1 Steps in a Call

The method by which a program transfers control to a subprogram is known as a **call**. In a Pascal program, for example, a subprogram may be a procedure, invoked with a procedure call statement, or a function, invoked by using the function name in an expression.

A calling sequence has three main functions.

- It must arrange to transfer program control to the subprogram for execution and back to the original procedure after completion.

- It must pass arguments to the subprogram and return results.
- It must establish an execution environment for the subprogram, and it must restore the original environment upon subprogram completion.

The precise steps in a call differs among different computers, but the steps in all calling sequences may be separated into five categories. In order of execution, they are:

Call Sequence Instructions executed by the calling procedure, or "caller," to prepare for transferring program control to a subprogram, or "callee." The sequence includes pushing procedure arguments onto the call frame stack, recording the current PC for a return address, and transferring control.

Entry Sequence Instructions executed by the subprogram to prepare for execution. Functions include saving registers that must be preserved after execution, allocating space for local variable storage, and adding unwind information to the call frame. This is sometimes referred to as **prologue code**.

Execution The body of the subprogram.

Exit Sequence Instructions that restore saved registers, put function results where the caller will find them, and pop local data off the stack. This is sometimes referred to as **epilogue code**.

Resume Sequence Program control returns to the caller.

The following two sections outline the details of these steps for the 680x0 family of processors and the Series 10000 processors. The sequences below describe only the most general case. If a call is internal to a compilation unit—the caller and callee procedure code are in the same source file—the compiler is free to optimize the argument passing conventions.

5.1.1 Steps in a Call for 680x0 Processors

On a 680x0 processor, the steps that take place during a subprogram call are as follows.

Call Sequence

1. If the argument list length is not a multiple of four bytes, the caller adjusts the stack pointer to ensure longword alignment for the callee.
2. The calling program pushes the subprogram's arguments onto the stack from right to left. The leftmost argument is pushed on last and is at the top of the stack. If the caller expects a returned value via a hidden argument, it must push onto the stack a memory address with sufficient space to hold the expected return value.
3. The caller pushes the return address onto the stack.

4. The caller jumps to the start of the subprogram.

   (The **JSR** or **BSR** machine instruction accomplishes steps 3 and 4 at once.)

Entry Sequence

5. If the callee routine references static data or external data or procedures, entry to the routine is through an **Entry Control Block** (ECB). The code within the ECB loads the address of the procedure's static data and external reference vector. (If the callee does not reference static external data and is only called from within its compilation unit, it does not need an ECB. The CALL instruction transfers control directly to the first callee instruction.) See Section 5.5 on ECBs for more information.

6. If the subprogram uses floating-point hardware, the called routine pushes the address plus 1 of a structure describing the saved floating-point registers.

7. The called routine pushes onto the stack the address of the caller's local stack area, so that each stack frame points back to the one that called it. This is accomplished by pushing the caller's SB register onto the stack and setting the callee's SB register to point to the saved caller's SB in the stack.

8. The called procedure allocates space on the stack for its local variables.

   (The **LINK** machine instruction accomplishes steps 7 and 8.)

9. The called routine pushes the registers that it might modify onto the stack. The called routine is responsible for preserving the state of all of the caller's registers except **D0**, **D1**, **A0**, **A1**, **FP0**, and **FP1**. (**FP8–FP15**, if present, are also not preserved.) The structure of the stack frame is detailed in Section 5.3.1.

Execution

10. The subprogram executes its code. If the subprogram is using the parameter list for output, the subprogram puts the output at those argument positions in the stack (or modifies locations pointed to by the parameter list).

Exit Sequence

11. The subprogram uses the stack copies to restore the preserved registers to their initial state (as they were at the start of the subprogram).

12. If the routine returns a value, the routine computes the value into register **D0** or **A0**. (Domain/Ada also uses **FP0** for floating-point return values.)

13. The called procedure loads the saved copy of the caller's pointer to its local data, restoring the SB register to its original value.

14. The called procedure pops its local data from the stack.

    (The **UNLK** machine instruction accomplishes steps 13 and 14 at once.)

15. The subprogram returns control to the calling procedure (using the **RTS** machine instruction).

Resume Sequence

16. The caller pops the subprogram's arguments off the stack. If any of these arguments contain or point to output from the subprogram, the caller may read that data.

5.1.2 Steps in a Call for Series 10000 Processors

This section describes the sequence of events that occur when a program invokes an external routine on a Series 10000 workstation.

Call Sequence

1. If the caller expects a function return value which will not be returned in a register, the caller must allocate space for the function result and pass its address as the first argument (in **.4**).

2. The caller of the routine loads the arguments into the argument passing registers (**.4–.9, .FS8–.FS19, .FD8–.FD18**). The caller stores any additional arguments in the argument block. If the caller expects a returned value via a hidden argument, it must load into the first argument register (**.4**) a memory address with sufficient space to hold the expected return value. This address is treated like the other arguments.

3. The caller loads the address of the callee routine into a register, usually **.0**. (The register is then used as the operand in the **CALL.S***x* instruction that follows.)

4. The caller loads the address of the return PC into **.22** (**.RETURN**) and transfers control to the callee. (The instruction **.RETURN = CALL.S***x* [*register*] performs these two steps.)

Entry Sequence

5. If the callee routine references static data or external data or procedures, entry to the routine is through an **Entry Control Block** (ECB). The code within the ECB loads the address of the procedure's static data and external reference vector. (If the callee does not reference static external data and is only called from within its compilation unit, it does not need an ECB. The CALL instruction transfers control directly to the first callee instruction.) See Section 5.5 on ECBs for more information.

6. The callee creates a new stack frame. If the routine is entered through an ECB, the instruction that creates the stack frame must immediately follow the branch instruction in the ECB. (It is said to be in the **shadow** of the branch instruction.) Small leaf routines can run without creating a stack frame because the return PC and arguments are kept in registers. The stack frame is described in Section 5.3.2.

7. If the routine is entered through an ECB, a **B.S***x* [*start_address*] instruction in the ECB causes the program to branch to the subroutine's procedure code.

8. If the called routine will change the contents of any preserved registers, the called routine must save the current contents of those registers in the stack. (See 'Preserved Registers' in Figure 5-2 and Figure 5-4.) The order that it saves the registers is low before high (**.11** before **.12**) and integer registers before floating-point registers. The .unwind section in the object file contains information on what registers are saved and the location of the save area relative to SF.

Execution

9. The called routine executes its code.

Exit Sequence

10. If the called routine is a function that returns an address or an integer value, it computes the return value into integer register **.0**. If the result is a single-precision floating-point value, it returns the result in **.FS0**.

    If the function result is not returned in a register, the address of the result is passed in register **.4**. (In Pascal, FORTRAN, and C, both the caller and callee must declare the function's return type. So, the caller and the callee can examine the function return type to determine whether the function value is returned in a register or if its address is passed in integer register **.4**.)

11. The called procedure restores the contents of the caller's preserved registers to their original values.

12. The called procedure returns control to the caller. The **B.S***x* **[.RETURN]** instruction accomplishes this task.

Resume Sequence

13. Execution resumes.

5.2 Register Usage

During a procedure or function call, the caller and callee use registers to manage the call stack frame and to return function values. Series 10000 workstations also use registers to pass arguments to subprograms. Called subprograms are also responsible for restoring some registers to the state they were in when the subprogram was invoked.

5.2.1 Register Usage for 680x0 Processors

Under Domain conventions, three of the 680x0 address registers have special functions:

A7 is the **stack pointer** (**SP**), which points to the top of the call stack. The stack grows from high addresses toward low addresses.

A6 is the **stack base** (**SB**), which points to a fixed position at the base of the stack frame of the currently active routine. Local variables and arguments are accessed relative to this register.

A5 is the **data base** (**DB**), which points to the start of the active routine's data section.

Procedures are required to preserve the contents of registers **D2–D7**, and **A2–A5**. **A6** is saved in the link field of the callee's stack frame. **A7**, the stack pointer, is not explicitly saved, but it is restored when the callee's call frame is popped from the stack before returning. Figure 5–1 illustrates the usage of the 680x0 registers.

Some Domain 680x0 workstations contain specialized floating-point hardware, either 6888x coprocessors or FPA or FPX accelerators. These accelerators provide the node with eight or sixteen floating-point registers, called **FP***x*. The registers may be for double-precision floating-point values (64 bits), or they may be large enough to hold extended precision values (80 bits). Procedures are required to preserve the contents of registers **FP2–FP7**. If they exist, the **FP8–FP15** registers are reserved for the math library, and are not preserved. Some workstations are equipped with the Performance Enhancement Board (PEB), a floating-point accelerator with a single accumulator rather than several registers. During a call, procedures are not required to preserve the state of the accumulator.

Registers **A0–A1**, **D0–D1**, and **FP0–FP1** are registers that the called procedure can use for any purpose and does not have to save. Small routines can often use only these registers and thus avoid saving and restoring registers. **D0** and **A0** are also used to return function results. (See Section 5.4.3 for more information on using these two registers in functions.)

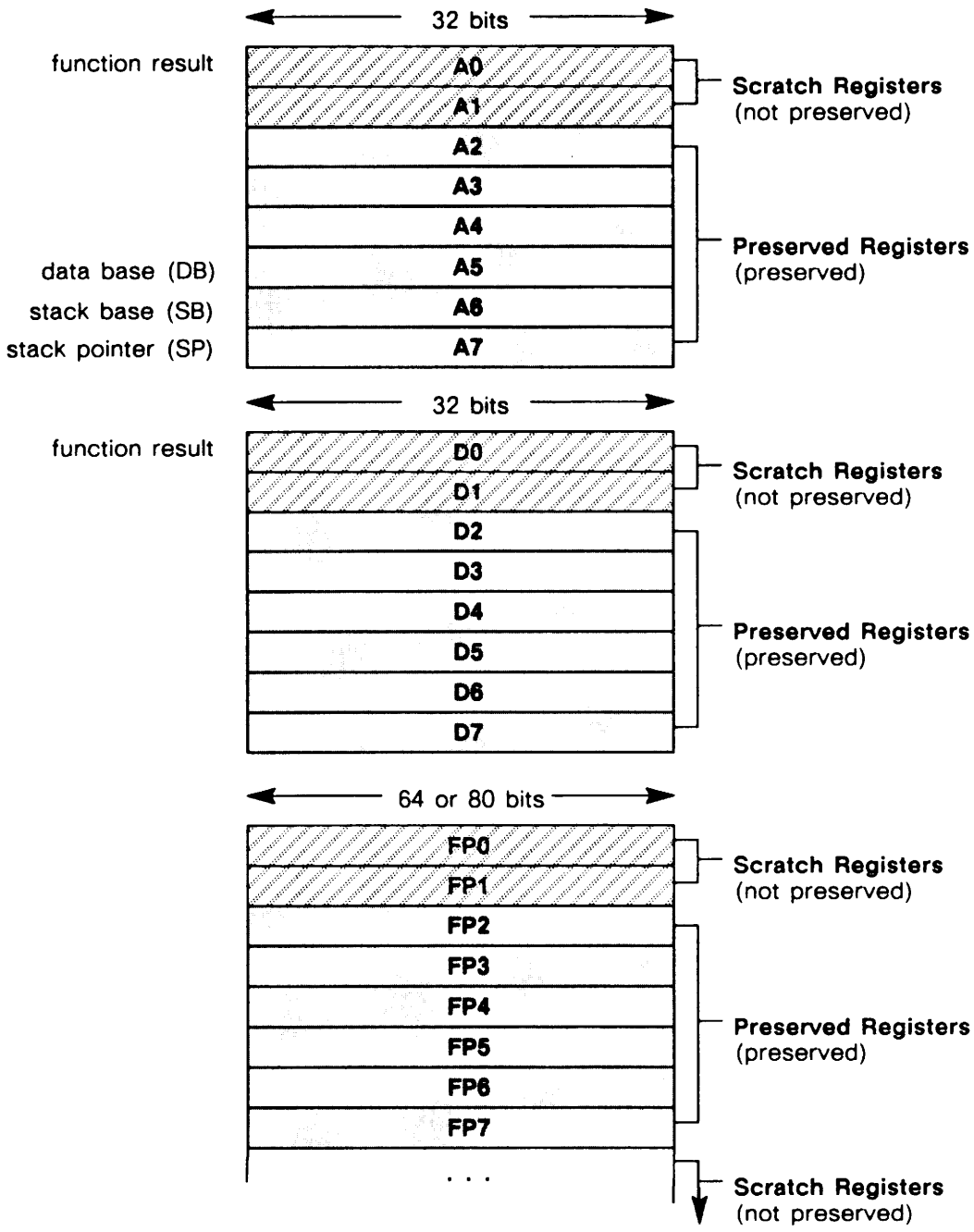


Figure 5-1. Register Usage on the 680x0

5.2.2 Register Usage for Series 10000 Processors

In contrast to the 680x0 calling conventions, which use two address registers to indicate the current state of the call stack, Series 10000 calling conventions use a *single* **stack frame**

pointer (SF, **.23**) to perform the functions of both the stack pointer (SP) and the stack base (SB).

The SF register points to the top of the procedure call stack and the stack frame of the current procedure. You can address all stack operands using SF; therefore, the Series 10000 workstation does not provide a separate SB.

In addition, Apollo's 680x0 conventions provide a separate **data base** register (DB) that points to the start of the active routine's data section. The Series 10000 conventions do not provide a *fixed* DB. Compiler code generators are free to use any register for DB. (They commonly use **.2** and **.21**.) This allows a leaf routine (a routine that makes no calls) to use any non-preserved register to point to its data section while an intermediate procedure can use a preserved register to point to its data section.

We detail in the following sections how Series 10000 calling conventions use integer registers and floating-point registers.

5.2.2.1 Integer Registers

The *PRISM* architecture contains 32 integer registers. The names of the registers are **.0**... **.31**. Each integer register is 32 bits wide. Figure 5-2 illustrates the integer registers and their usage. In addition, Section 5.4.3.3 contains examples of C function calls that illustrate the use of registers.

Registers **.0** to **.3** are scratch registers that the callee (the called procedure) does not have to save. Procedures can use these registers for data that does not need to be preserved across calls. Two of these registers perform special functions:

.0	Contains a function result that is an address, an integer value, or a **record**, **struct**, or **union** whose size is less than or equal to 4 bytes.
.1	Holds the restart address in lock sequences.

Registers **.4** to **.9** serve as argument passing registers. It is not necessary for the callee to preserve argument registers. See Section 5.4 for more information.

Registers **.10** to **.23** are preserved registers. The callee is responsible for preserving these registers across calls. Two of these registers perform special functions defined by software convention:

.22	Contains the return address on entry to an external procedure. (Internal procedures can use any register for the return address as long as the caller and callee agree.)
.23	Points to the top of the stack (SF). The stack grows from high addresses to low addresses.

Registers **.24** to **.31** are the system registers; they are write-protected in user state. Registers **.24** to **.27** are defined by software convention; registers **.28** to **.31** are hardware de-

fined. Register **.30** contains the current PC and **.31** is the **null** register. See the *Series 10000 Technical Reference Library, Volume 1* for a complete description of these registers.

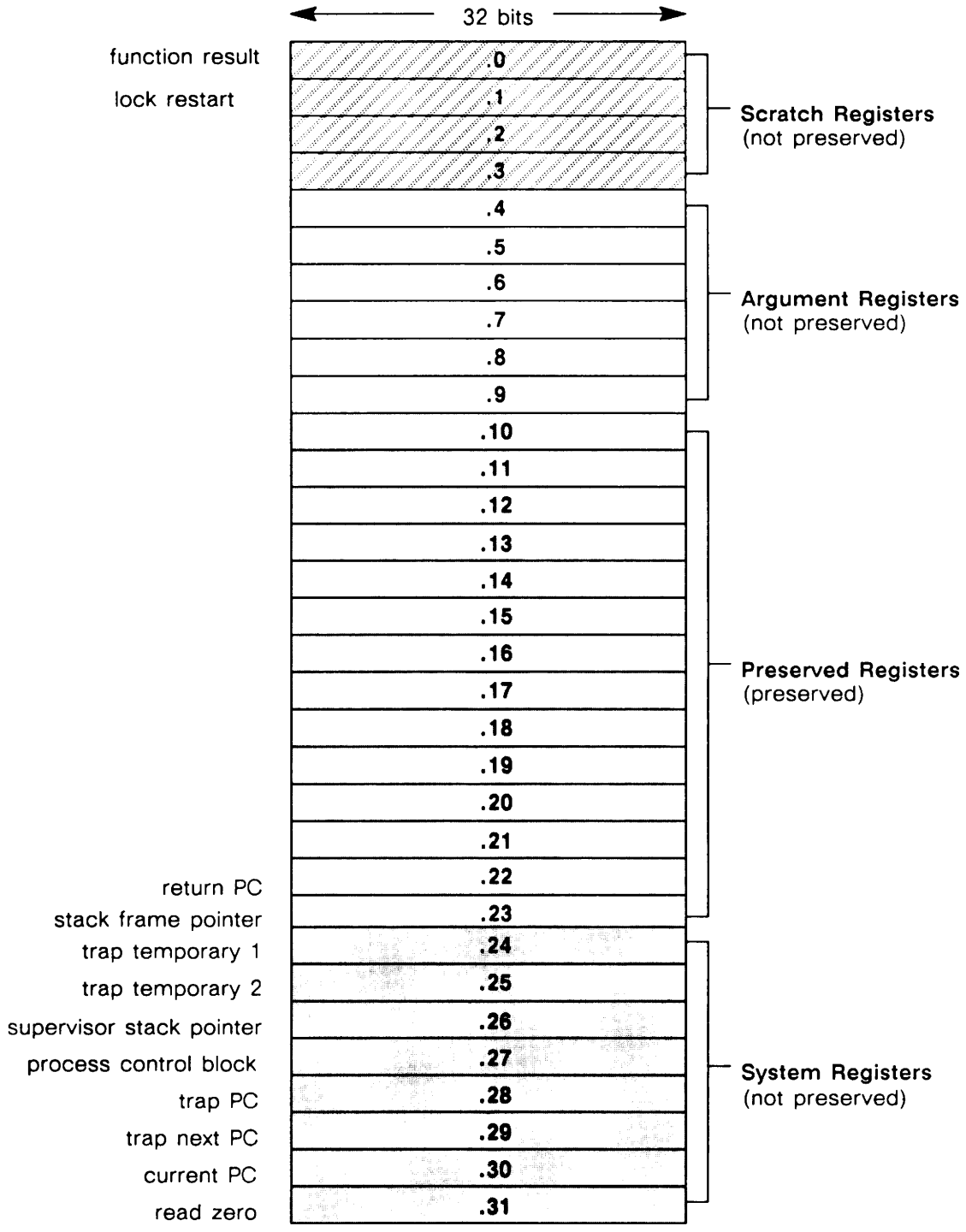


Figure 5-2. Integer Register Usage

5.2.2.2 Floating-Point Registers

The *PRISM* architecture provides 64 single-precision registers overlaying 32 double-precision registers for floating-point values. The 64 single-precision registers (**.FS0**, **.FS1**, **.FS2**, ... **.FS63**) are 32 bits wide, and the 32 double-precision registers (**.FD0**, **.FD2**, **.FD4**, ... **.FD62**, even numbers only) are 64 bits wide.

Two adjacent single-precision registers that are an even-odd pair correspond to each double-precision register. For example, single-precision registers **.FS0** and **.FS1** overlay the double-precision register **.FD0**. (See Figure 5-3.) This allows you to use a double-precision floating-point LOAD or STORE operation to LOAD or STORE two adjacent single-precision floating-point registers in a single operation (or a LOAD or STORE of two single-precision registers to LOAD or STORE an unaligned double-precision value.)

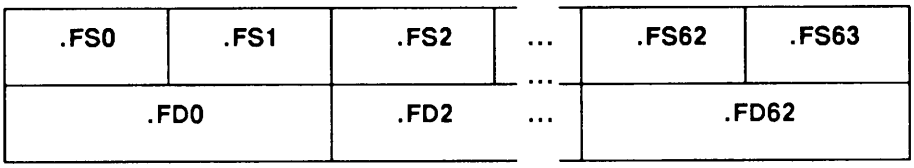


Figure 5-3. Register Correspondence

Figure 5-4 illustrates the double-precision floating-point registers and their usage.

Registers **.FS0** to **.FS3** (**.FD0** to **.FD2**) and **.FS20** to **.FS51** (**.FD20** to **.FD50**) are scratch registers. Procedures can use these registers for data that does not need to be preserved across calls. Some of these registers perform special functions:

.FS0	Contains single-precision function results.
.FD0	Contains double-precision function results.

Registers **.FS4** to **.FS7** (**.FD4** to **.FD6**) and **.FS52** to **.FS63** (**.FD52** to **.FD62**) are preserved registers. The callee is responsible for preserving these registers across calls.

Registers **.FS8** to **.FS19** (**.FD8** to **.FD18**) serve as argument passing registers. It is not necessary for the callee to preserve argument registers. See Section 5.4 for more information about argument registers.

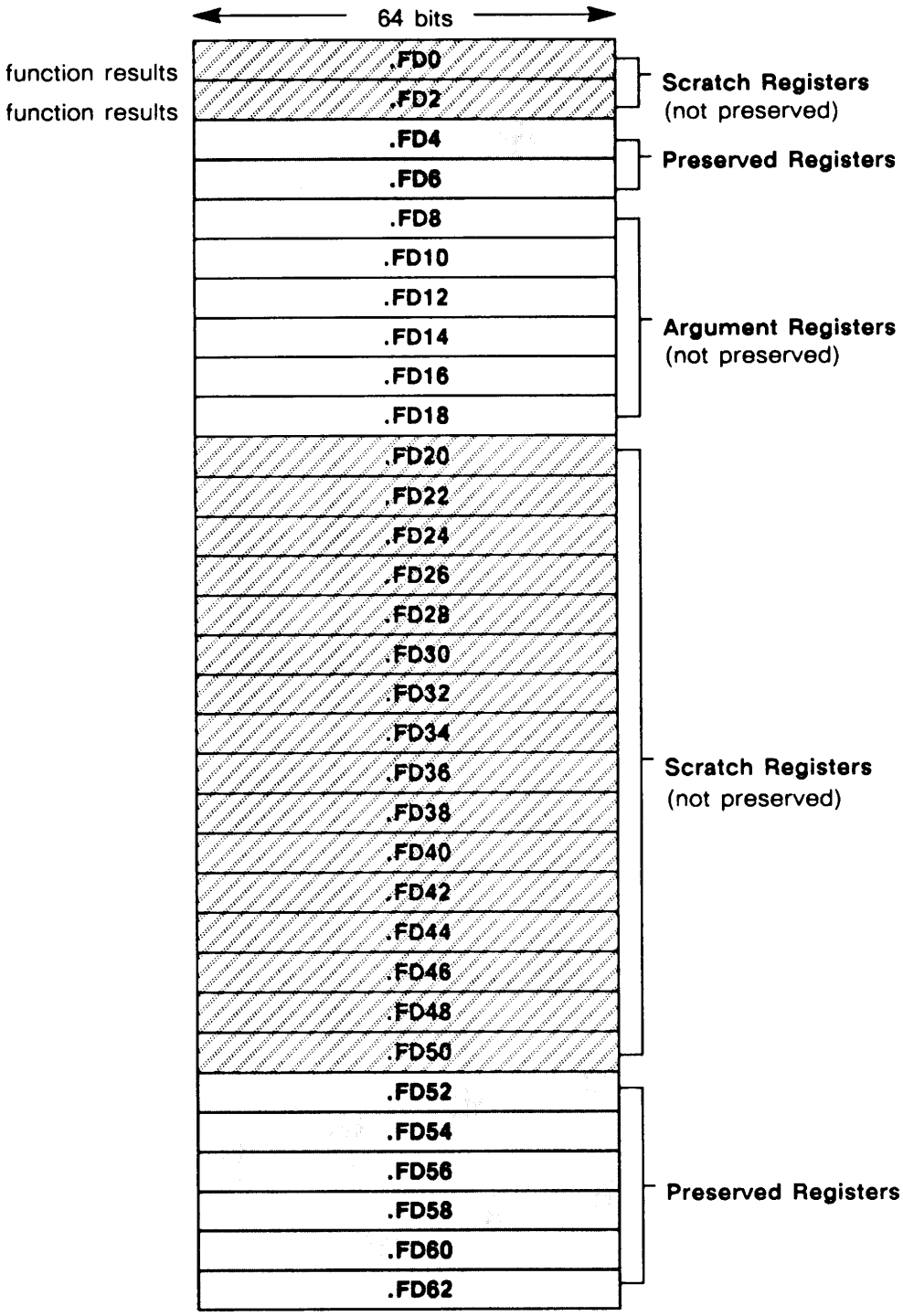


Figure 5-4. Floating-Point Register Usage

5.3 Stack Frame

The **stack frame**, or **call frame**, of a procedure is a record of the execution context of that procedure. It contains the arguments with which the procedure was called, the state of the PC before invocation, registers to be restored when execution completes, and the location of the previous frame on the stack. The stack frame may include other information, depending on the node type.

5.3.1 Stack Frame for 680x0 Processors

Figure 5–5 shows the structure of a 680x0 procedure stack frame in the most general case. Not all of the fields are present in every actual stack frame. The fields are in order, with the bottom field, the arguments, pushed onto the stack first.

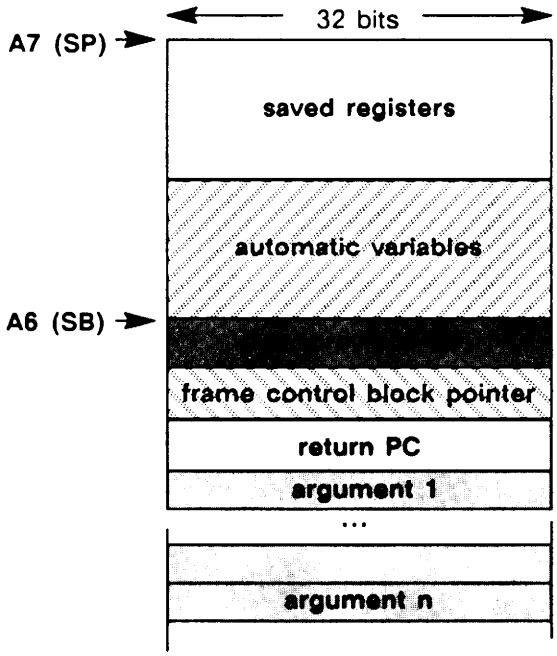


Figure 5–5. 680x0 Stack Frame Format

Arguments The caller pushes arguments onto the stack. They are pushed in order from right to left. If the call will use a hidden argument, the caller pushes it onto the stack last. This is sometimes referred to as the zeroth argument, and is between the return PC and argument 1.

Return PC The JSR or BSR machine instructions push this address onto the stack and transfer control to the routine.

Frame Control Block Pointers

If any preserved floating-point registers are changed by the callee, the routine must push the address of a Frame Control Block (FCB) onto the stack. If no preserved registers are affected by the callee, no FCB is necessary. FCBs provide supplementary information for unwinding the stack. If an FCB is present, its address *plus 1* is pushed onto the stack. FCBs must begin at even addresses, so the address plus 1 is odd. This distinguishes an FCB pointer from the return address, which is always even.

The FCB itself is a 64-bit object consisting of a 16-bit value describing the kind of floating-point hardware, a 16-bit register mask whose bits are set corresponding to saved registers, and a 32-bit displacement from the stack base (SB), indicating the location of the saved floating-point registers.

The value representing the kind of floating-point hardware is a 16-bit integer equal to one for 6888x coprocessors, two for DNx60 nodes, and three for FPA-equipped nodes.

The register mask is a mask of the registers saved. A one indicates that the corresponding register is saved. For floating-point hardware with 16 registers, bit 0 corresponds to **FP15**. For hardware with eight registers, bit 0 corresponds to **FP7**.

The magnitude of the displacement value is equal to the size of the routine's automatic storage plus the size of the saved integer registers plus the size of the saved floating-point registers. Its sign is negative.

Previous SB The Stack Base (SB) register (**A6**) points to this field, which contains the caller's SB register value. Thus, it provides a link back to the caller's stack frame.

Automatic Variables

This field provides storage for the procedure's local and temporary variables.

Saved Registers If the procedure changes any of registers **A2** to **A5**, **D2** to **D7** or **FP2** to **FP7**, the caller's values are saved here. Note that **A6** is saved in the link field. **A7** is not explicitly saved; it is restored when the callee's parts of the stack frame are popped before returning.

5.3.1.1 Stack Growth

On Apollo's 680x0-based systems, there are two registers that contain stack information: the stack base (SB) and the stack pointer (SP). During the execution of a procedure, SB points to a fixed position in the stack frame of the currently active routine. SP, which points to the top of the call stack, is free to move as the caller pushes and pops arguments and local stack variables during calls.

5.3.2 Stack Frame for Series 10000 Processors

Figure 5-6 illustrates the format of a procedure **stack frame** on the Series 10000 workstation.

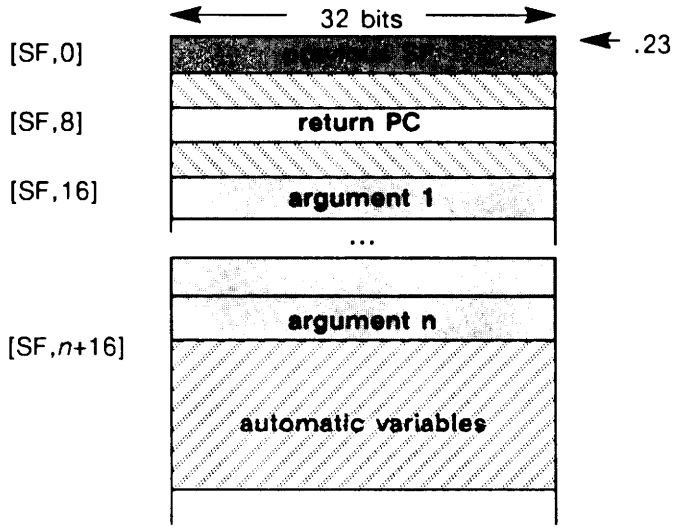


Figure 5-6. Series 10000 Stack Frame Format

Here, we describe the fields within the stack frame.

Previous SF Contains the address of the previous stack frame. The location of the previous SF is [SF,0]. Register **.23** points here.

Return PC Contains the return address of the routine that creates the stack frame. The return PC is not always saved on the stack. Since register **.22** usually contains this address, a leaf procedure—a procedure that makes no external calls—can leave the value in the register. If the return register must be saved on the stack, it is saved at [SF,8]. The empty spaces at [SF,4] and [SF,12], are for any other registers that may need to be preserved.

Argument block Contains arguments passed to routines that are called by the current routine. The size of the argument block must be large enough to hold the largest argument list of any called routine, and it must be at least 24 bytes long. Arguments that correspond to the first 24 bytes of the caller's argument block are passed in registers, so the callee can use these bytes for any purpose. The remaining bytes in the argument block contain arguments not passed in integer or floating-point registers. The location of the argument block is [SF,16]. See Section 5.4.2 for more information. If the call will use a hidden argument, the caller pushes it onto the stack last. This is sometimes referred to as the zeroth argument, and will appear (if used) in register **.4**.

Automatic Variables

Contains the procedure's local stack variables. The offset from SF depends on the size of the argument block (*n*).

The routine that creates the stack frame can use the longwords at [SF,4] and [SF,12] to save preserved registers. Preserved registers other than those saved at [SF,4] and [SF,12] are usually saved in the automatic variable area.

5.3.2.1 Stack Growth

In contrast to the 680x0, the Series 10000 workstation has a single stack frame pointer (SF) that must *always* point to a *complete* stack frame. A stack frame cannot be created incrementally, it must be allocated all at once. This means that you cannot arbitrarily push and pop items on the stack. If a procedure requires temporary storage, it must create a stack frame large enough to accomodate all the temporary storage the callee may need. For protection against asynchronous faults (such as a quit signal), you must create stack frames with an **atomic** operation. That is, you must use a single instruction that pushes a new stack frame and links the new stack frame to the previous stack frame. The following examples show how this is done in Series 10000 assembly language (**prasm**).

This example works if the frame size is less than or equal to 128 bytes:

```
.1                  = SF
[--SF,frame_size]   = .1
```

Note that an instruction such as

```
[--SF,frame_size]   = SF
```

is indeterminate (unless the page containing the instruction is in wired-down memory. You should avoid using this instruction because if a page fault occurs on the store of SF, the system will store the *updated* contents of SF.

This example works if the frame size is greater than 128 bytes but less than or equal to 256 KB:

```
.0                  = #-(frame_size/8)          ; an immediate value
.1                  = SF
[++SF,.0*8]         = .1
```

NOTE: An additional restriction to modifying SF is that SF must always contain an address that is a multiple of eight; that is, it must point to a quadword boundary. This means that you should round all stack frame sizes to a multiple of eight. (This is so you can always reference temporary stack variables with known alignments.)

5.4 Argument Passing

In a call of an external procedure, a caller must provide arguments to the subprogram. The caller and callee must agree on the argument passing strategy. On a 680x0 workstation, a caller simply pushes the callee's arguments onto the call frame stack. On a Series 10000 workstation, a caller uses registers to pass the first few arguments, but leaves blank space for those arguments on the stack. The remaining arguments are simply passed on the stack. On both nodes, software convention dictates that a caller pushes a subprogram's arguments onto the stack from right to left, putting the first argument in the calling sequence at the lowest address.

Under some circumstances, such as when a C function must return a structure or union that will not fit into a register, the calling program must include a **hidden argument** in the argument list. This is an argument that is on the stack but is not visible to the caller routine. Commonly, a hidden argument is pushed last onto the stack (and is occasionally referred to as the "zeroth" argument) and contains the address of a memory location where there is room enough to store the expected return value. However, FORTRAN uses hidden arguments which may be pushed onto the stack first.

5.4.1 Argument Passing for 680x0 Processors

The Domain 680x0 nodes use the stack to pass arguments from one procedure to another. The argument section of the call frame simply consists of the value or address of each argument pushed onto the stack in order from right to left. The last argument in the call becomes the first argument pushed onto the stack and the first argument in the call is pushed last.

5.4.2 Argument Passing for Series 10000 Processors

By software convention, the Series 10000 workstation uses six integer registers and six double-precision floating-point registers to pass arguments. When there are more arguments than can be passed in the argument registers, the caller passes the arguments in an argument block at a fixed offset—[SF,16]—in its stack frame. This section details argument registers and the argument block.

5.4.2.1 Argument Registers

Integer registers **.4**–**.9** contain arguments passed by reference and non-floating-point arguments passed by value. Register **.4** contains the first argument, register **.5** contains the second argument, and so on.

Floating-point registers **.FD8** to **.FD18** (**.FS8** to **.FS19**) contain double-precision (single-precision) floating-point arguments passed by value. The floating-point argument registers

can pass up to 12 single-precision floating-point arguments or six double-precision floating-point arguments.

When passing arguments in registers, the registers that pass the arguments follow strictly ascending order. For example, if you pass a 4-byte floating-point value, an 8-byte floating-point value, and another 4-byte floating-point value (in that order), the arguments are passed in **.FS8**, **.FD10**, and **.FS12** not in **.FS8**, **.FD10**, and **.FS9**.

The Series 10000 workstation uses registers to pass arguments until it reaches the end of the argument list or until it uses all the argument passing registers. A routine can pass an argument in a register even if a prior argument was not passed in a register. This occurs when the routine uses all of the integer registers but floating-point argument registers remain or when the routine uses all of the floating-point argument registers but general argument registers remain. The routine passes any remaining arguments in the argument block.

5.4.2.2 The Argument Block

The caller is responsible for allocating the argument block in its stack frame—at an offset of [SF,16]—and allocating enough space (a minimum of 6 longwords) to pass all the arguments of its longest call, including those passed in registers. The argument block must be large enough to hold the largest argument list of any of the called routines. Uninitialized storage areas for all the arguments to the call, regardless of their type, are laid out in order in the argument block. The caller reserves storage for the first argument to the call at [SF,16]. The storage locations of the remaining arguments depend on their types, sizes, and the source language. With the exception of arguments of fewer than 32 bits (8-bit or 16-bit integers, for example) passed by value, there is *no* alignment padding in the argument block. Therefore, argument block storage for double-precision arguments is not necessarily naturally aligned.

The callee can use the uninitialized argument block storage that corresponds to arguments passed in registers for any purpose; there is always a minimum of 24 bytes available in the caller's stack frame. (Note that this means that a small leaf routine need not allocate a stack frame.) The remainder of the argument block contains arguments passed to callees that are not passed in integer or floating-point registers. This occurs when there are more arguments than can be passed in registers. (See the following sections for examples of argument block layout.)

5.4.3 Language Argument Passing Conventions

A program or procedure may pass arguments to another procedure by **value** (the actual value of the argument) or by **reference** (the address of the argument). Each language's definition specifies how arguments are passed. The Series 10000 argument passing conventions imitate those used on 680x0 workstations. This section describes the Domain higher-level language argument passing conventions, and their implementation by the 680x0 and Series 10000 processors.

5.4.3.1 Pascal

By default, Pascal passes all arguments of externally called routines by reference regardless of the parameter mode (**in**, **out**, **in out**, or **var**). However, you can pass arguments by value rather than by reference if you use the **val_param** option in the procedure or function heading. In addition, Pascal has a **C_param** routine option that tells the compiler to return function results in register **D0** (on 680x0 workstations), and to pass all record data types by value rather than by reference. This facilitates cross-language calling with C and FORTRAN. See the *Domain Pascal Language Reference* for details on these two routine options.

> **NOTE:** Any Pascal procedure that calls or is called by a C program **must** use the **C_param** option to pass by value any single-precision or double-precision floating-point, RECORD or struct, simple datum, or pointer.

In an external call, even arguments with call-by-value semantics (which is the default) are actually passed by reference. For example, consider the following code fragment:

```
PROCEDURE dump (size: integer)
          .
          .
          .
          IF size > maxsize THEN size := maxsize;
          .
          .
          .
```

Pascal semantics require that the change to **size** not affect the value of the caller's actual parameter. The code that the Pascal compiler generates for this call takes care of this, by making a local copy of the argument that can be modified.

You can specify that a procedure's arguments be passed by value with the **val_param** option (see the *Domain Pascal Language Reference* for its format). Under this option, arguments that are four bytes or less in size and are not arrays are passed by value. Arguments larger than four bytes and all arrays are passed by reference.

Internal routines can be called only within the same compilation unit in which they are defined. Since the compiler knows all the calls to the routine, it can optimize argument passing without regard to the argument passing conventions. Currently however, internal routines are treated the same as **val_param** routines.

680x0 Pascal Conventions

On the 680x0 workstations, the caller procedure widens irregularly sized arguments which are to be passed by value before pushing them onto the stack. Arguments of less than 16 bits are widened to 16 bits, and arguments of between 16 and 32 bits are widened to 32 bits. Arguments of more than 32 bits are widened to an even number of bytes.

Pascal returns values from functions through registers. Integer and character values are returned in register **D0**, and pointers are returned in register **A0**. All other values are returned in the memory location specified by an address passed by a hidden argument

Series 10000 Pascal Conventions

In Pascal integer registers **.4** to **.9** can contain pointers to arguments passed by reference or non-floating-point arguments passed by value. Floating-point registers **.FD8** to **.FD18** (**.FS8** to **.FS19**) contain floating-point arguments passed by value. The argument block contains any additional arguments as well as arguments that don't fit into registers.

Consider the following Pascal procedure declaration:

```
procedure arg_layout (IN int_1,int_2,int_3,int_4,int_5,int_6,int_7:integer32;
                      IN OUT real_1,real_2:real); EXTERN;
```

Suppose the call to procedure **arg_layout** is:

```
arg_layout(int_1,int_2,int_3,int_4,int_5,int_6,int_7,real_1,real_2);
```

Figure 5-7 illustrates how the caller passes some arguments in registers, while passing other arguments in the argument block (of the caller).

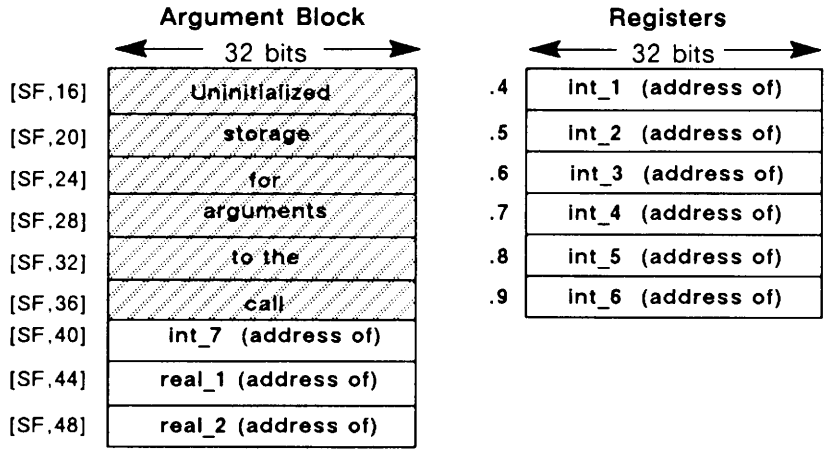


Figure 5-7. Argument Passing in Pascal

In the preceding example, the caller passes all the arguments (regardless of their types) by reference. Therefore, the caller passes the addresses of the first six arguments in registers **.4** through **.9**., and passes the addresses of the remaining three arguments in the argument block (beginning at [SF,40]).

Consider the same procedure declaration, but this time we use the **val_param** option to tell the compiler to pass the arguments by value rather than by reference:

```
procedure arg_layout (IN int_1,int_2,int_3,int_4,int_5,int_6,int_7:integer32;
                      IN real_1,real_2:real); VAL_PARAM; EXTERN;
```

Once again, the call to procedure **arg_layout** is:

```
arg_layout(int_1,int_2,int_3,int_4,int_5,int_6,int_7,real_1,real_2);
```

Figure 5-8 illustrates how the caller passes most of the arguments in registers, and passes only the argument **int_7** in the argument block (of the caller).

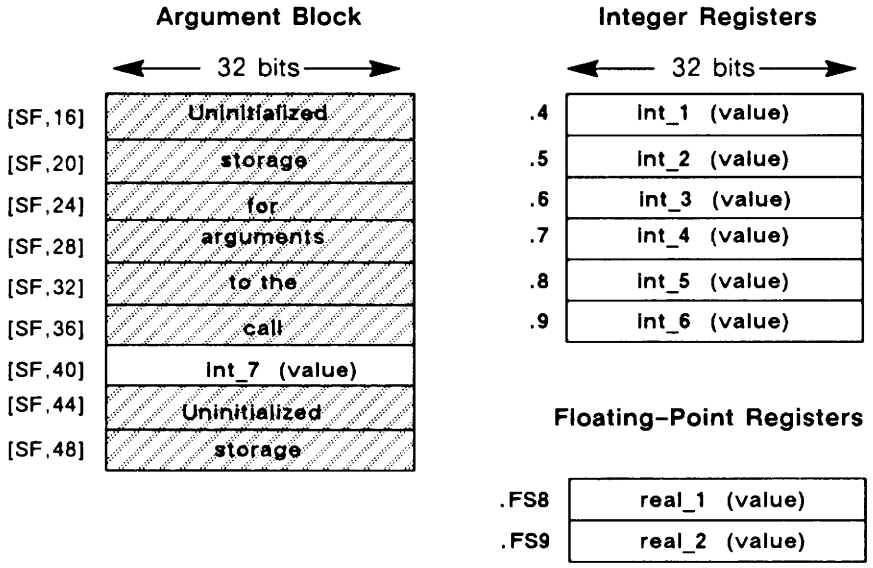


Figure 5-8. Argument Passing in Pascal Using the Val_Param Option

When using the **val_param** option, the caller passes all **IN** (and default) arguments by value. Therefore, the caller passes the values of the first six integer arguments in registers **.4** through **.9**. In addition, the caller passes the value of the seventh integer argument in the argument block at [SF,40], and passes the last two argument to the call—since they are floating-point arguments—in **.FS8** and **.FS9**. (Note that space is reserved in the argument block for the last two arguments to the call.)

With internal routines, which are called only from within the compilation unit that defines them, the compiler can optimize argument passing without using the calling conventions.

5.4.3.2 FORTRAN

In FORTRAN, all arguments are always passed by reference, with the single exception of FORTRAN CHARACTER strings. When passing a CHARACTER string, FORTRAN passes not only the address of the string, but also its length, and length arguments are sometimes passed by value.

The default (no **-uc**) calling conventions are:

- A CHARACTER string occupies eight bytes in the argument list. Four bytes hold the address of the string, and four bytes hold the address of the length of the string, a short integer. The length arguments of *all* the strings in the argument list are pushed *first* onto the stack, from right to left, before any other arguments.

- If this is a FUNCTION returning a CHARACTER string, the address of the returned string is passed as a hidden argument, pushed *last* onto the stack. Its length is a short integer passed by reference, pushed onto the stack *before* all the explicit arguments, but *after* any length arguments of strings in the argument list.

- All other parameters are passed by reference in succeeding registers, or in the argument block in the caller's stack frame.

Under the **-uc** (UNIX compatiblity) switch, the following conventions apply:

- A CHARACTER string occupies eight bytes in the argument list. Four bytes hold the address of the string, and four bytes hold the value of the length of the string, a long integer. The length arguments of *all* the strings in the argument list are pushed *first* onto the stack, from right to left, before any other arguments.

- If this is a FUNCTION returning a CHARACTER string, the address of the returned string is passed as a hidden argument, pushed *last* onto the stack. Its length is a long integer passed by value, pushed onto the stack *immediately before* the string address, *after* all the explicit arguments in the argument list.

- All other parameters are passed by reference in succeeding registers, or in the argument block in the caller's stack frame.

The **-uc** option's effect on FORTRAN CHARACTER length arguments is the same on a Series 10000 as on a 680x0 workstation.

680x0 FORTRAN Conventions

With the exception of the CHARACTER length hidden arguments detailed above, FORTRAN argument passing conventions are quite simple. The caller pushes the address of each argument onto the call stack beginning from the end of the subroutine call. The leftmost argument is the last argument pushed.

When returning function results, integers go in **D0**. Everything else is passed in a memory location specified by a hidden argument pushed last onto the argument stack.

Series 10000 FORTRAN Conventions

Apart from the exception noted above, FORTRAN passes all arguments by reference. Integer registers **.4** to **.9** contain the addresses of the first six arguments. The argument block contains the addresses of any additional arguments.

Since FORTRAN always passes arguments by reference, the argument block layout and what arguments are passed in registers are fairly easy to understand.

Suppose this is the call to the subroutine **arg_layout**:

```
int*4     int_1, int_2
real*4    real_1, real_2
real*8    double_1, double_2
character letter
.
.
.
call arg_layout (int_1, int_2, real_1, real_2, double_1, double_2, letter)
.
.
.
```

Figure 5-9 illustrates how the caller passes most of the arguments to the call in registers, and passes only the argument **letter** in the argument block (of the caller).

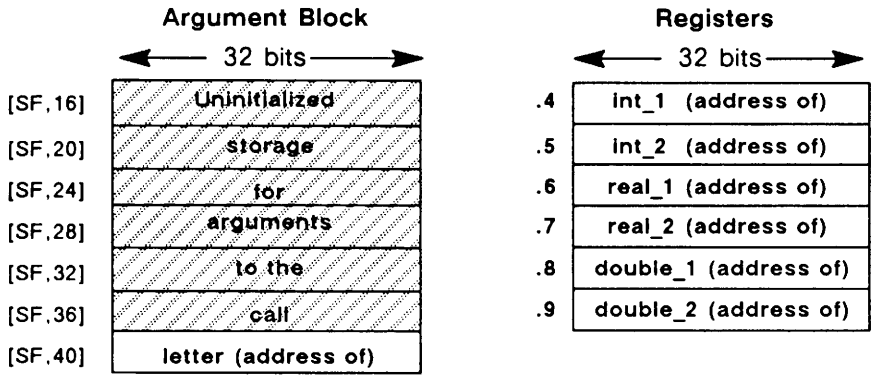


Figure 5-9. Argument Passing in FORTRAN

The caller passes all of the arguments—with the exception of **letter**—in registers. Register .4 contains the address of **int_1**, register .5 contains the address of **int_2** and so on. **Letter** is passed in the argument block at the location [SF,40]. Since the first 24 bytes of argument are passed in registers, the corresponding bytes in the argument block are never initialized with the addresses of the arguments. Therefore, the callee is free to use the first 24 bytes of the argument block for any purpose.

FORTRAN COMPLEX function results are returned in memory. A pointer to the destination COMPLEX is passed in **.4**, similar to the "C returning a struct" or "Pascal returning a RECORD" conventions. Note that this has the effect of pushing all function arguments

into the next register. This change makes the Series 10000 conventions compatible with the 680x0 convention with respect to FORTRAN COMPLEX functions and cross-language calling.

5.4.3.3 C Language

Traditional C passes scalar arguments by value and arrays by reference, but Domain/C incorporates ANSI and C++ features that allow you to pass arguments by value or by reference depending on the presence of **function prototypes**. Function prototypes are function declarations that include type information for the arguments. These prototypes combined with C++ style **reference variables** give C a way to pass arguments by reference.

> **NOTE:** The ANSI C standard makes obsolete the old-style function declarations. Programs with prototypes will execute faster on both 680x0 and Series 10000 nodes.

The following example demonstrates the traditional way to define a function in C:

```
int example(point_1, count)
char *point_1;
int count;
{
  /* function body */
}
```

Here, both **point_1** and **count** are passed by value. The **point_1** variable is actually a pointer to a **char**, but it is passed by value. The **count** variable may be changed during execution of **example**, but the changed value will not be returned to the caller.

By contrast, we can define the same function arguments in Domain/C with function prototypes, as follows:

```
int example(char *point_1, int &count)
{
  /* function body */
}
```

The **&** preceding **count** tells the compiler to pass **count** by reference. Now, if the count variable is changed during execution of example, the changed value will be returned to the caller.

C without Function Protoypes

In the case where there is no function prototype for a called function (or where there is a '...' prototype), C passes all arguments (except arrays) by value. Furthermore, C specifies that certain conversions are automatically made when passing a value as an argument. Table 5-1 lists the argument type conversions in C without function prototypes.

Table 5–1. Argument Type Conversions in C without Function Prototypes

Actual Argument Type	Converted to
char, short, enum	**int** (= **long** in Domain/C)
unsigned char, unsigned short	**unsigned int** (= **unsigned long**)
float	**double**
array of *T* (any data type)	pointer to *T*
function returning *T*	pointer to function returning *T*

The argument passing conventions for C without prototypes are influenced by C functions that expect a variable number of arguments (for example, the **printf** function) and access the arguments as if they were stored contiguously in the argument block. C without prototypes requires that arguments passed by value (with the exception of arrays) be widened to **int** and **double**. Since no padding is allowed in the argument block, the layout of arguments in the argument block can cause unaligned **double** arguments (arguments that are longword aligned but not quadword aligned).

680x0 C without Function Prototypes

C without prototypes passes all integer and pointer arguments in four bytes, float and double arguments in eight bytes. Shorter data types are widened according to the rules in Table 5–1. Structs passed by value are widened to a multiple of four bytes. Integers, floats, pointers, and structures and unions of fewer than four bytes are returned in **D0**. Longer data types, including double-precision values and structures and unions of more than four bytes, are passed in a memory location specified by a hidden argument pushed last onto the argument stack.

Series 10000 C without Function Prototypes

C without prototypes passes integer and pointer arguments in four bytes, float and double arguments in eight bytes. Callers without prototypes pass doubles in the next double-precison floating-point registers *and* in the next two integer argument registers; eight bytes are set aside in the argument block. Callers without prototypes pass floats by widening them to doubles and following the rules for doubles. Integer registers **.4** to **.9** contain the first six longwords of argument. Note that this *does not* mean the first six arguments—two integer registers can pass a **double** argument. For example, suppose that integer registers **.5** and **.6** contains a **double** argument. To access the argument, you would **MOVE** the contents of the two integer registers into a double-precision **FP** register as the following examples illustrate:

```
.FS6.I    = .5
.FS7.I    = .6
```

Then you can access the argument by using the following double-precision floating-point register term (remember that each even-odd single-precision register pair overlays a double-precision register):

```
.FD6
```

In the following example, **f** is widened to double and passed in (**.4**,**.5**) *and* in **.FD8**, **d** is passed in (**.6**,**.7**) *and* in **.FD10**, and **i** is passed in **.8**.

```
int     i;
float   f;
double  d;
void    ralph();
          .
          .
          .
    ralph( f, d, i );
```

On a Series 10000 machine, all callees, with or without prototypes, expect double-precision arguments in the next available double-precision argument register, and skip the next two integer argument registers. For example, in the following code, **f** is expected in **.FD8** (**.4** and **.5** are skipped), **d** is expected in **.FD10** (**.6** and **.7** are skipped) and **i** is expected in **.8**.

```
void ralph( f, d, i )
double d, f;
int i;
{
   .
   .
   .
}
```

If the code in the previous example was used to call this function, **f** will also be in **.4** and **.5**, and **d** will also be in **.6** and **.7**. However, this is irrelevant to the purposes of the subprogram. The conventions call for this redundancy to allow programmers to mix functions with and without prototypes.

> **NOTE:** C functions without prototypes convert the types of many of their arguments. Table 5-1 lists these conversions. The redundancy in register use for double-precision arguments means that the callee will always be able to find its arguments. It does *not* guarantee that the callee will know what type they will be.

Callees that use VARARGS and "..." prototypes expect all arguments in integer argument registers (or in the overflow area in the caller's argument block).

We next present an example of how a C function call without prototypes passes arguments.

The function and argument declarations and the function call are as follows.

```
int arg_layout(int_1, double_1, int_2, real_1, real_2, letter )
int int_1, int_2;
double double_1, double_2;
float real_1, real_2;
char letter;
   .
   .
   .
   answer = arg_layout (int_1, double_1, int_2, double_2, real_1, real_2, letter )
   .
   .
   .
```

Figure 5–10 illustrates how the caller passes some arguments in registers, while passing other arguments in the argument block (of the caller).

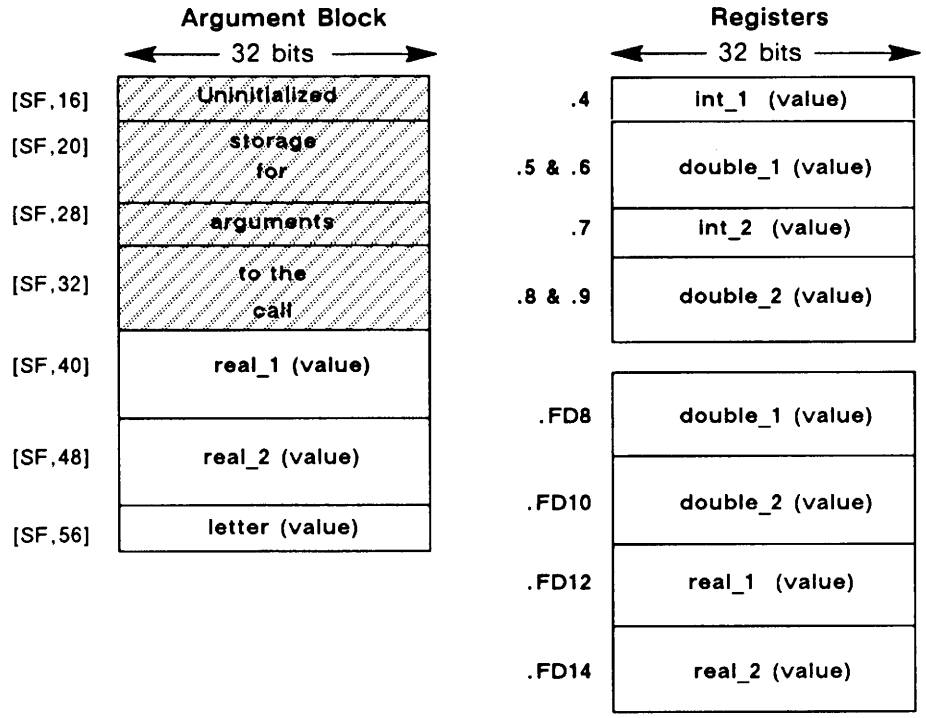


Figure 5–10. Argument Passing in C without Function Prototypes

The caller passes the first four arguments (**int_1**, **double_1**, **int_2**, and **double_2**) in registers. Register **.4** contains the value of **int_1** and register **.7** contains the value of **int_2**. Since **real_1** and **real_2** are double values, the caller passes each argument in two integer registers; **double_1** in registers **.5** and **.6** and in **.FD8**, and **double_2** in registers **.8** and **.9** and in **.FD10**.

The caller passes the remaining arguments in the argument block at the locations listed. Remembering that C automatically converts a float to a double when passing the argument as a value, you'll notice that the caller passes the fifth and sixth arguments to the call, **real_1** and **real_2**, in the argument block as 8-byte values. These two arguments are also passed in **.FD12** and **.FD14**. The argument **letter** is converted to an **int** and passed in the argument block as a 4-byte value.

Since the caller passes the first 24 bytes of argument in registers, the caller never initializes the corresponding bytes in the argument block with the argument values. Therefore, the callee is free to use these 24 bytes for any purpose. In C without function prototypes, argument block storage that corresponds to arguments passed in registers is always 24 bytes.

C with Function Prototypes

In the case where there are function prototypes, C can pass arguments by value or by reference depending on how the parameter is declared. However, there is *no* automatic conversion when passing a value as an argument. C with function prototypes passes **char** and **short** arguments in registers (one per register) and allocates these arguments four bytes each in the argument block. C with function prototypes uses floating-point registers to pass floating-point values.

> **NOTE:** The caller and the callee must agree on the use of function prototypes. If the caller uses prototypes, the callee must be prepared to accept prototype-style parameters. Similarly, if the callee uses prototypes, the caller must pass prototype-style parameters.

680x0 C with Function Prototypes

C with function prototypes does not have to pad or convert its arguments, except that 8-bit values are widened to two bytes. Integers, floats, pointers, and structures and unions of fewer than four bytes are returned in **D0**. Longer data types, including double-precision values and structures and unions of more than four bytes, are passed in a memory location specified by a hidden argument pushed last onto the argument stack.

Series 10000 C with Function Prototypes

We next present an example of how arguments are passed for a C function that uses function prototypes and reference variables. Consider this function definition:

```
int arg_layout (int int_1, int &int_2, double double_1, double &double_2, float
                real_1, float &real_2, char letter)

/* & preceding variable names tells compiler to pass the argument by reference */
{
/* function body */
}
```

Invoke the function with this call:

```
answer = arg_layout (int_1, int_2, double_1, double_2, real_1, real_2, letter )
```

Figure 5–11 illustrates how the caller passes all the arguments in registers.

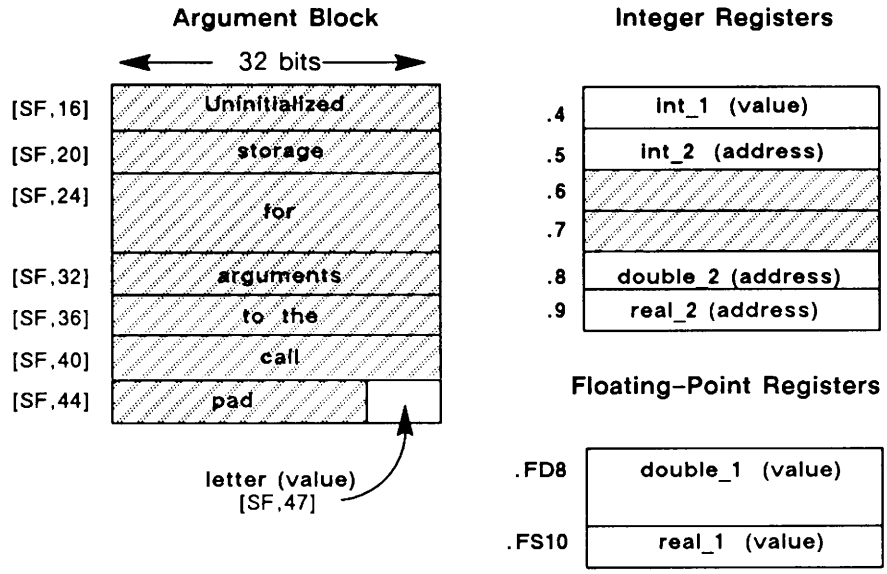


Figure 5–11. Argument Passing in C Using Function Prototypes and Reference Variables

In this example, the caller passes all arguments to the call in registers, including floating-point values. Therefore, the caller is free to use the uninitialized argument block storage for any purpose.

Callers with prototypes pass doubles in the next double-precision floating-point register and skip the next two integer argument registers; eight bytes are set aside in the argument block.

5.4.3.4 Function Results

Function results of four bytes or less are returned in registers. Function results larger than four bytes long must be returned via a "hidden" argument. The caller of a function that returns a result larger than 4 bytes pushes an additional argument onto the stack. This additional argument is the address of the location where the result is to be stored. The argument logically precedes all others (that is, it is the last one pushed onto the stack).

680x0 Function Results

Function results are returned in a register if they are four bytes or less in size. In C and FORTRAN, these function results are returned in D0. In Pascal, the result is returned in A0 if it is a pointer, and in D0 if not. For cross-language compatibility, the Pascal **d0_return** option causes pointer results to be returned in both A0 and D0.

Function results larger than four bytes are returned via a hidden argument, which contains the address of the function result. On the 680x0 workstations, the hidden argument is immediately adjacent to the return PC in the stack frame, at the head of the argument block. See Section 5.3.1 for more information about the stack frame.

Series 10000 Function Results

Integer register **.0** returns function results that are non-floating-point scalar data types (such as pointers and integers), as well as records, structs, or unions that are four bytes or less.

If a function returns a non-floating-point result that is larger than four bytes, the caller of the function must allocate storage space for the function's return value and then pass the address of the return value in the first argument register (**.4**).

Floating-point register **.FS0** returns single-precision floating-point function results, and register **.FD0** returns double-precision floating-point function results.

FORTRAN returns a **COMPLEX** function result in **.FS0** (the real part) and **.FS1** (the imaginary part) and a **DOUBLE COMPLEX** function result in **.FD0** and **.FD2**.

5.4.3.5 Library Routines

All arguments of system library routines are generally passed by reference. This allows the routine to be called from Pascal, FORTRAN, and C. Alternatively, arguments can be explicitly declared as pointers in the C header file. Passing all arguments by reference is sometimes referred to as **standard calling conventions**.

If you write your own routines intended to be called from multiple languages, beware of data types (such as Boolean) that do not have direct analogs in all languages.

5.5 Entry Control Blocks

An **entry control block** (ECB) is a block of code prepended to a routine. The code in the ECB is executed when control is transferred from a caller routine to a callee subprogram. ECBs are associated with routines; there is one ECB per routine rather than one ECB per routine call. Entry to the routine is through the ECB. The ECB contains code that loads the address of the area that contains the static data and external references.

ECBs are part of the routine's **prologue** code. Prologue code is the set of instructions that compilers generate at the beginning of a routine to set up a stack frame and, if necessary, a data frame pointer. **Epilogue** code is the set of instructions generated to return from the routine.

5.5.1 680x0 Entry Control Blocks

Compilers for 680x0 workstations create ECBs only for position-independent code (compiled with the **-pic** option) which references static read/write data or external data or procedures.

5.5.2 Series 10000 Entry Control Blocks

Series 10000 compilers generate ECBs for every externally visible routine that references static read/write data or external data or procedures.

Series 10000 compilers generate this ECB if the routine creates a stack frame whose frame_size is 128 bytes or less:

```
          DATA
data_frame
          EQU               *

*         ECB is part of routine's data section
          EXPORT.P          routine_name
routine_name
          .0                = ac_text_start    ; load procedure code
                                               ; address
          .2                = #data_frame      ; load data_frame pointer
          .1                = SF               ; SF is .23
          B.SA              [.0]               ; branch to procedure
                                               ; code
          [--SF,frame_size] = .1               ; create stack frame,save
                                               ; and preserve SF pointer
                                               ; frame_size must be a
                                               ; multiple of 8 bytes
ac_text_start
          DATA              text_start         ; address constant for
          .                                    ; start of routine
          .
          .
```

If the routine creates a stack frame that is larger than 128 bytes but less than 256KB, Series 10000 compilers generate the following ECB:

```
        DATA

data_frame
        EQU             *

*       ECB is part of routine's data section
        EXPORT.P        routine_name

routine_name
        .0              = ac_text_start     ; load procedure code
                                            ; address
        .2              = #data_frame       ; load data_frame pointer
        .1              = SF                ; SF is .23
        B.SA            [.0]                ; branch to procedure
                                            ; code
        .0              = #-((frame_size+7)/8)
                                            ; frame_size must be
                                            ; multiple of 8 bytes
ac_text_start
        DATA            text_start

        PROC
text_start                                  ; begin .text section
         [++SF,.0*8]    = .1                ; create stack frame
        .
        .
        .
```

Note that the location of the stack push instruction is different in the two examples. There are four possibilities for the location of the stack push instruction:

- The ECB contains the stack push instruction. If this is the case, the stack push instruction must be in the shadow of the branch instruction in the ECB. (This is a requirement of the stack unwinding procedure invoked during fault handling: if the PC appears in the procedure text of a called routine, then that routine must have already pushed its stack frame. In contrast, if the PC is still in the ECB for the routine, then the routine must not have pushed a stack frame.)

- The routine has an ECB but the stack push instruction is in the procedure code rather than the ECB.

- The routine creates a stack frame but has no ECB. So the stack push instruction is in the procedure code.

- The routine has an ECB but doesn't create a stack frame. So there is no stack push instruction.

Figure 5–12 contains an example of the external call mechanism and the prologue and epilogue code it generates.

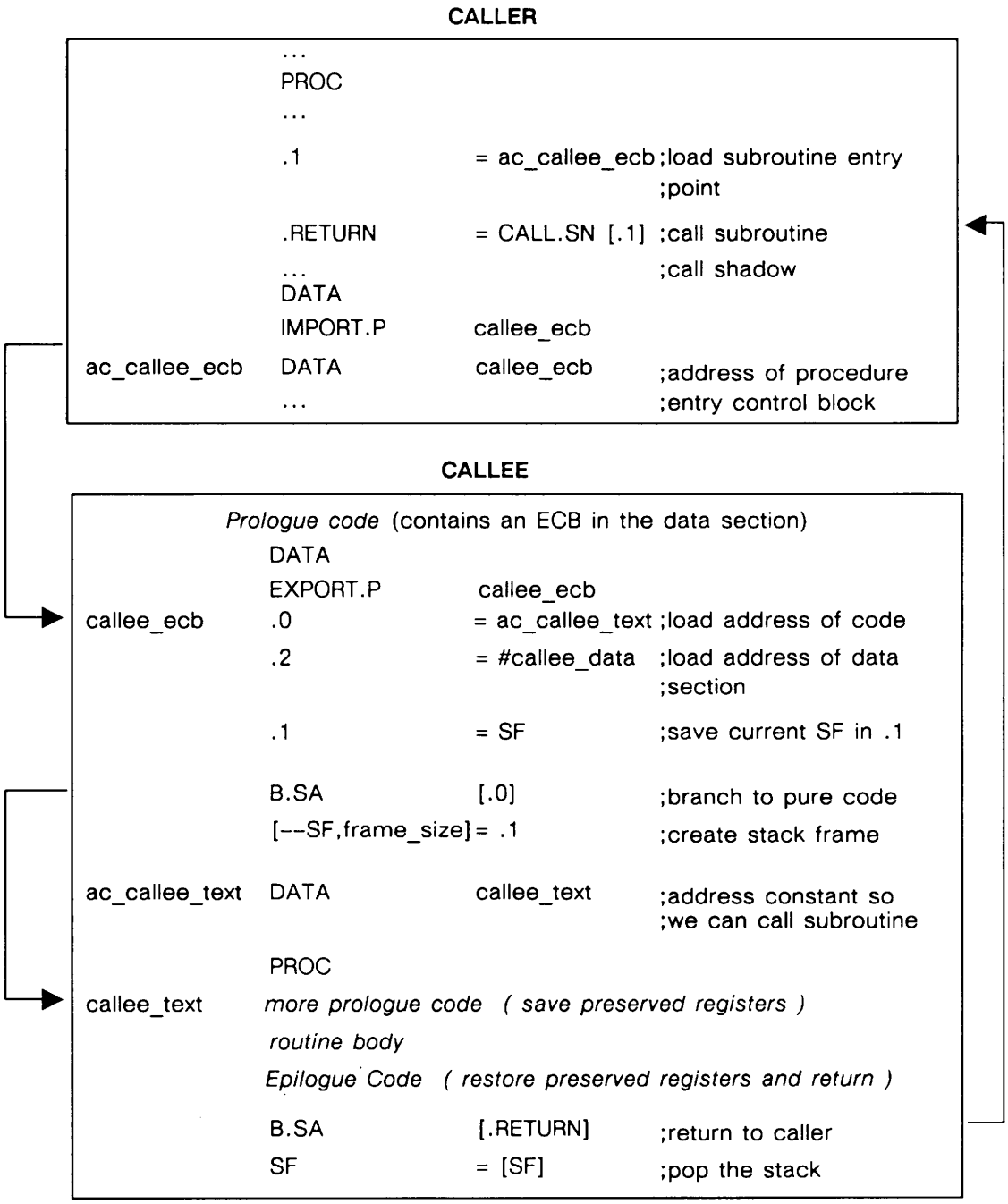


Figure 5–12. External Call Mechanism Displaying Prologue and Epilogue Code

Chapter 6

COFF:
Common Object File Format

This chapter describes the Apollo implementation of the Common Object File Format (COFF). COFF is a standard object file format instituted by AT&T. Many vendors of UNIX System V systems, including Apollo, use COFF as their object file format. Since SR10, the object files created by all Domain compilers and assemblers are COFF files. Knowledge of the object file format is essential for writing compilers to generate COFF files and for writing debuggers or dis-assemblers to read COFF files.

Chapter 3 provides an overview of the parts of a COFF file and how COFF files are treated by the Domain linkers and loader. You should understand the material in Chapter 3 before reading this chapter.

The first section of this chapter contains an overview of the component parts of a COFF file. The subsequent sections detail the function and format of each individual part.

As you read this chapter, you may find the commands **coffdump** (in BSD and Aegis), **dump** (in SysV), and **dstdump** (in all three environments) useful. The **coffdump** and **dump** commands display selected parts of an existing COFF file. Options, documented in the **coffdump** and **dump** man pages, determine which part(s) will be displayed. The **-v** (verbose) modifier, which displays information in symbolic rather than numeric form, may be especially useful.

The **dstdump** command displays interpretations of selected parts of the debugging sections **.unwind**, **.blocks**, **.symbols**, and **.lines**. (The **coffdump** and **dump** commands will not interpret the information from these sections.) Use **coffdump**, **dump**, or **dstdump** to view any existing COFF file.

When using the tables in this chapter, remember that all Apollo machines are **big-endian**, that is, the most significant byte in a multi-byte field is the one with the lowest address. Also, within a byte, bits are numbered from the least significant (bit 0) to the most (bit 7).

6.1 Review of the Parts of a COFF File

Figure 6-1 illustrates the parts of the AT&T COFF template as described in Chapter 3. In addition to the required parts specified by the template, all Apollo COFF files have an optional header, between the file header and the table of section headers, and a section called **.unwind**.

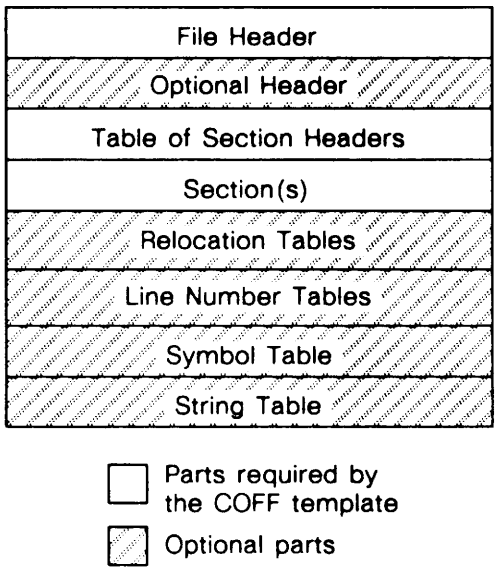


Figure 6-1. Parts of a COFF file

These are the parts of the COFF file shown in Figure 6-1:

- **Sections** contain the information central to the purpose of the COFF file. The COFF template defines text sections, initialized data sections, and uninitialized data sections; a compiler or linker may define additional types of sections. A program's machine instructions are typically stored in text sections. The number of sections allowed in a COFF file is virtually unlimited, and the COFF file template requires no specific structure for additional section types. The compiler, therefore, is free to create several sections to store the program, and to create one or more additional sections to store any other information about the file, as all Domain compilers do.

The other component parts merely contain information to support the sections.

- The **file header** and the **optional header** are records containing some information about the file as a whole.

- A **section header** specifies the size, location, and type of a section; each section in the file must be described by a section header.

- **Relocation tables** identify address references that the linker and/or loader must change as they reposition sections.
- The **line number table** maps source code line numbers to the addresses of their corresponding instructions.
- The **symbol table** stores information about program symbols, such as their data types and values.
- The **string table** stores the character strings to which other parts of the COFF file refer.

As you read this chapter, you may find the header files that define the COFF file parts useful. Table 6-1 lists these header files.

Table 6-1. Header Files Defining the Parts of a COFF File

COFF File Part	Header File(s)
File header	**/usr/include/filehdr.h**
Optional header	**/usr/include/aouthdr.h**
Section headers	**/usr/include/scnhdr.h**
.sri section	**/usr/include/sri.h**
.mir section	**/usr/include/mir.h**
.inlib section	**/usr/include/inlib.h**
.rwdi section	**/usr/include/scndata.h**
.unwind section	**/usr/include/apollo/unwind.h**
.blocks section	**/usr/include/apollo/dst.h**
.symbols section	**/usr/include/apollo/dst.h** **/usr/include/apollo/isp.h**
.lines section	**/usr/include/apollo/dst.h**
Relocation tables	**/usr/include/reloc.h**
Line number tables	**/usr/include/linenum.h**
Symbol table	**/usr/include/syms.h** **/usr/include/storclass.h**

NOTE: The header files define many values that Apollo COFF files don't use. In general, this chapter describes only the values that Apollo COFF files use.

6.2 File Header

Every COFF file has a file header. The file header contains general information about the object file and its contents. From the file header, you can find

- The processor type on which the file is executable.
- The number of sections in the file.
- When the file was created.
- The location of the COFF symbol table.
- The number of entries in the symbol table.
- The size of the optional header.
- Whether the file is executable.
- Whether the file includes a line number table.
- Whether the COFF symbol table includes descriptions of local symbols.
- Whether the file has been stripped of unnecessary symbols and their strings.
- Whether the file contains breakpoints inserted by a debugger.
- Whether the file contains absolute code or position-independent code (in Apollo COFF files only).

The **coffdump** or **dump** command with the **-f** option displays the file header.

Table 6-2 shows the format of the file header, and Table 6-3 lists the flags used in it.

Table 6-2. Format of the File Header

Byte Offset	Field Name	Value
0-1	**f_magic**	One of the following octal values, identifying the processor type on which the file is executable: 0627 680x0 processors 0624 Series 10000 processors
2-3	**f_nscns**	The number of sections in the COFF file.
4-7	**f_timdat**	When the file was created, expressed in **time_$clockh_t** format. See note in Section 6.5.9.3 for information about this format.
8-11	**f_symptr**	Byte offset from the beginning of the COFF file to the beginning of the COFF symbol table.
12-15	**f_nsyms**	The number of entries in the symbol table (including auxiliary entries).
16-17	**f_opthdr**	The number of bytes in the optional header.
18-19	**f_flags**	A set of flags providing additional information about the file. Table 6-3 describes each of the flags that Apollo COFF files use. **f_flags** is the logical OR of all the applicable flags.

Table 6-3. Flags in the File Header

Hex Value	Flag Name	Meaning
0x01	**F_RELFLG**	Some relocation information has been stripped from the file.
0x02	**F_EXEC**	The file is executable.
0x04	**F_LNNO**	Line number information has been stripped from the file.
0x08	**F_LSYMS**	Local symbols are not included in the symbol table.
0x2000	**F_BREAKPOINT**	The file contains breakpoints inserted by a debugger. This flag is an Apollo extension to the COFF standard.
0x4000	**F_STRIPPED**	The file has been stripped of unnecessary symbols and their strings. This flag is an Apollo extension to the COFF standard.
0x8000	**F_PIC**	The loader can change the file's virtual addresses: the file contains position-independent code, all of its relocation information, and its symbol table. This flag is an Apollo extension to the COFF standard.

6.3 Optional Header

A COFF file may contain a second header called the **optional header**, which is typically used for system-specific information. In Apollo COFF files, this header is required. Programs which do not require the information in the optional header may skip over it by using the optional header size, **f_opthdr**, found in the file header. From the optional header in Apollo COFF files, you can find

- The version of the Apollo COFF format used by the program file. (The original Apollo COFF format, instituted at SR10, is version 1).
- The version of the program stored in the COFF file.
- The size of the portion of the object file that the loader will map, read-only, into memory. (This value includes all text sections and the file header, optional header, and section headers.)
- The combined size of all the initialized data sections in the file.
- The size of the uninitialized data in the file.
- The virtual address of the program's entry point.
- The virtual address of the first text section.

- The virtual address of the first data section.
- The location within the COFF file of the **.sri** section.
- The location within the COFF file of the **.inlib** section.

The **coffdump** or **dump** command with the **-o** option displays the optional header.

Table 6-4 shows the format of the optional header created by all Domain compilers, linkers, and assemblers.

Table 6-4. Format of the Apollo Optional Header

Byte Offset	Field Name	Value
0-1	**vformat**	An integer representing the version of the Apollo COFF format used by this program.
2-3	**vstamp**	An integer representing the version of the program stored in this file (as specified by linker options) or 0 (if no version is specified).
4-7	**tsize**	The total number of bytes that the loader will map, read-only, into memory. This figure includes all text sections and the file header, optional header, and section headers.
8-11	**dsize**	The total number of bytes in the initialized data section(s).
12-15	**bsize**	The total number of bytes in the uninitialized data section(s).
16-19	**entry**	The virtual address of the program's first instruction.
20-23	**text_start**	The virtual address of the beginning of the first text section.
24-27	**data_start**	The virtual address of the beginning of the first data section.
28-31	**o_sri**	Byte offset from the beginning of the COFF file to the beginning of the **.sri** section, or 0 if there is no **.sri** section.
32-35	**o_inlib**	Byte offset from the beginning of the COFF file to the beginning of the **.inlib** section, or 0 if there is no **.inlib** section.
36-43	**vid**	Not used.

Many other UNIX vendors use the optional header format shown in Table 6-5, commonly known as the "a.out header". It includes a "magic number" in place of the information about the version of the COFF format, and it omits the information about the location of the **.sri** and **.inlib** sections. The Apollo header format is similar enough to this format that tools expecting an a.out header might work on Apollo files.

Table 6-5. Format of the Optional Header Used by Many Other UNIX Vendors

Byte Offset	Field Name	Value
0-1	**magic**	One of the following octal values, indicating how the file should be executed: 0407 The text section is not write-protected and can't be shared, and the data section is contiguous with the text section. 0410 The text section is write-protected and the data section immediately follows the text section. 0413 Text and data sections are aligned with **a.out**.
2-3	**vstamp**	An integer representing the version of the compiler or linker that created this object file.
4-7	**tsize**	The total number of bytes that the loader will map, read-only, into memory.
8-11	**dsize**	The total number of bytes in the initialized data section(s).
12-15	**bsize**	The total number of bytes in the uninitialized data section(s).
16-19	**entry**	The virtual address of the program's first instruction.
20-23	**text_start**	Byte offset, from the beginning of the COFF file, to the beginning of the first text section.
24-27	**data_start**	Byte offset, from the beginning of the COFF file, to the beginning of the first data section.

6.4 Section Headers

Following the file header(s), every COFF file contains a table of section headers. Every section has a section header. From a section header, you can find

- The section's name
- The location of the section, in virtual address space and within the COFF file
- The size of the section
- The location of the section's relocation information
- The location of the section's line number information
- The number of relocation entries for the section
- The number of line number entries for the section
- The section's type

The **coffdump** or **dump** command with the **-h** option displays the section headers.

Table 6-6 shows the format of a section header. Table 6-7 lists the flags used in a Section header, and Table 6-8 details the use of two of those flags.

Table 6-6. Format of a Section Header

Byte Offset	Field Name	Value
0-7	**_n**	If the section name contains 8 or fewer characters, this field is named **s_name** and contains the section name (padded with trailing zeros, if necessary). In Apollo COFF files, if the name is longer than 8 characters, this field is composed of two fields: **_n_zeroes** (4 bytes of zeros) followed by **_n_offset** (the byte offset, from the beginning of the COFF string table, to the section name).
8-11	**s_paddr**	The virtual address of the beginning of the section, or 0 if the section has no virtual address. (The value of this field is identical to the value of **s_vaddr**.)
12-15	**s_vaddr**	The virtual address of the beginning of the section, or 0 if the section has no virtual address. (The value of this field is identical to the value of **s_paddr**.)
16-19	**s_size**	The amount of space that the loader must allocate for the section in bytes.
20-23	**s_scnptr**	Byte offset from the beginning of the COFF file to the beginning of the section or 0 if the section has no contents.
24-27	**s_relptr**	Byte offset from the beginning of the COFF file to the section's relocation entries.
28-31	**s_lnnoptr**	Byte offset from the beginning of the COFF file to the section's line number entries.
32-33	**s_nreloc**	The number of relocation entries for the section, to a maximum of 65535. If the section has more than 65535 relocation entries, **s_nreloc** is 65535, and the section's first relocation entry stores the actual number of relocation entries in its **r_vaddr** field and the enumerated value **r_nreloc** in its **r_type** field. This is an Apollo extension to the COFF standard.
34-35	**s_nlnno**	The number of line number entries for the section.
36-39	**s_flags**	A set of flags identifying the section's type. Table 6-7 describes each of the flags that Apollo COFF files use. **s_flags** is the logical OR of all the applicable flags.

Table 6-7. Flags in a Section Header

Hex Value	Flag Name	Meaning
0x20	**STYP_TEXT**	The section is a text section.
0x40	**STYP_DATA**	The section is a data section.
0x80	**STYP_BSS**	The loader will fill the space allocated for this section with zeros. If the **STYP_COMPRESSED** flag is also set, the section's contents are stored in compressed form; the loader will write the compressed data into the section. Otherwise, the section is an uninitialized data section.
0x200	**STYP_INFO**	The section contains information about the object file that neither the linkers nor the loader needs. The section does not have a virtual address.
0x00010000	**STYP_RELOCATED_NOT_LOADED***	The section has a virtual address, but the loader will not load it.
0x00020000	**STYP_DEBUG***	The section contains information for Domain/DDE (the debugger).
0x00040000	**STYP_OVERLAY***	The linkers will overlay all sections that have the same name as this section.
0x00800000	**STYP_ZERO***	Before copying data into this section, the loader will fill the space allocated for the section with zeros.
0x02000000	**STYP_INSTALLED***	If this file is used as an installed library, programs that have access to the library will have access to this section. By default, Domain compilers and **ld** set this flag. Options to **bind** turn this flag on or off. A program can access this section by setting the **STYP_LOOK_INSTALLED** flag on a section of the same name.
* This flag is an Apollo extension to the COFF standard.		

(Continued)

Table 6-7. Flags in a Section Header (Cont.)

Hex Value	Flag Name	Meaning
0x02000000	**STYP_INSTALLED***	If this file is used as an installed library, programs that have access to the library will have access to this section. By default, Domain compilers and **ld** set this flag. Options to **bind** turn this flag on or off. A program can access this section by setting the **STYP_LOOK_INSTALLED** flag on a section of the same name.
0x04000000	**STYP_LOOK _INSTALLED***	The loader will look for a section in an installed library to use in place of this section's contents. If the loader finds an installed library section that has the same name as this section and has the **STYP_INSTALLED** flag set, the loader will use the installed section's contents. Otherwise, the loader will allocate space for this section. Linker options turn this flag on or off.
0x08000000	**STYP_SECALIGN1***	This flag and the **STYP_SECALIGN2** flag form a 2-byte field that indicates the section's alignment, as described in Table 6-8.
0x10000000	**STYP_SECALIGN2***	This flag and the **STYP_SECALIGN1** flag form a 2-byte field that indicates the section's alignment, as described in Table 6-8.
0x20000000	**STYP _COMPRESSED***	The section's contents are stored in compressed form. If this flag is set, the **STYP_BSS** flag must also be set.
* This flag is an Apollo extension to the COFF standard.		

Table 6-8. **STYP_SECALIGN1** *and* **STYP_SECALIGN2**

STYP_SECALIGN1	STYP_SECALIGN2	Meaning
0 (not set)	0 (not set)	The section is long-aligned (aligned on a 4-byte boundary).
0 (not set)	1 (set)	The section is quad-aligned (aligned on an 8-byte boundary).
1 (set)	0 (not set)	Undefined.
1 (set)	1 (set)	The section is page-aligned (aligned on a page boundary).

6.5 Sections

Following the table of section headers are the sections themselves. The COFF template defines three types of sections: **text sections, intialized data sections**, and **uninitialized data sections**. These three types of sections store the program's machine code and its data. They may store other information, as well. Apollo COFF files, for example, store the information required to unwind the stack in a text section named **.unwind**. (The **.unwind** section is described in Section 6.5.8.)

The COFF template requires that every COFF file have at least one section. Beyond that restriction, a COFF file may contain any number of text, initialized data, and/or uninitialized data sections. Sections 6.5.1 and 6.5.2 describe text and data sections.

The COFF template allows the compiler to define additional types of sections, as well. Domain compilers, linkers, and assemblers take advantage of this flexibility and define several additional types of sections. Sections 6.5.3 through 6.5.11 describe all of these sections.

A section is composed of a **section header** and **section contents**. The section header is stored in the table of section headers, which we described in Section 6.4. A flag in the section header identifies the section type.

Every section has a header. Not every section has contents, however. Uninitialized data sections, for example, represent uninitialized C global data. These data are not stored explicitly in a COFF file: they are represented by a section header only. The section header specifies the size of the data. The loader allocates the specified amount of space for the section when it loads the COFF file.

The COFF template does not specify the order in which sections must appear. The Domain loader, however, does require that all text sections appear first, followed by all initialized data sections, followed by all uninitialized data sections. Additional sections may follow in any order.

> **NOTE:** Some utilities that read COFF files assume a specific section order. Many simple object files contain one text section, one data section, and one uninitialized data section, in that order. Some utilities, therefore, assume that each COFF file will have exactly those sections, in that order. This is a dangerous assumption. A typical simple Apollo COFF file, for example, contains two text sections, not one. Do not assume which sections a COFF file will have or how they will be ordered. Read the section headers to determine what types of sections are included in the file and in what order.

6.5.1 Text Sections

The COFF template defines a text section as read-only material that will be loaded into memory. Typically, text sections are used to store a program's machine instructions. Compilers may also store other read-only information in text sections.

By default, a compiler creates a text section named **.text** to store the program's machine code. Some languages allow the programmer to create additional text sections and to store the machine code for specified portions of source code in those sections. Domain/C, for example, provides the **#module** and **#section** preprocessor directives for creating new text (and data) sections. (See the *Domain/C Language Reference* for more information.) Programmers may wish to create their own sections in order to minimize paging by ensuring that specific instructions are grouped together in the object file.

To load a text section, the Domain loader maps the section from the COFF file into address space. Because text sections are read-only, they cannot contain data values that will change as the program runs. For the same reason, they cannot contain address references that are relocatable at load time. This means that programs using position-independent code cannot store explicit address references in text sections.

6.5.2 Data Sections

Because text sections are read-only, program data, which change as the program runs, are stored in separate read-write sections known as data sections. Programs that use position-independent code also store any explicit address references in data sections.

By default, a compiler creates a data section named **.data** to store the program's static data and relocatable references. Some languages allow the programmer to create additional data sections and to store specific data in those sections. A FORTRAN COMMON block, for example, specifies a separate data section for all the data in the block. Programmers may wish to create their own sections in order to minimize paging, by ensuring that specific data are grouped together in the object file.

6.5.2.1 Compressed Data Sections

Domain compiler options allow you to store data sections in a compressed form. The combination of the **STYP_BSS** and **STYP_COMPRESSED** flags in the section header identify a section as representing compressed data.

A compressed data section has a section header, but no section contents. The section header specifies the amount of space the loader must allocate for the section. The section data are stored, in compressed form, in a section named **.rwdi**. A single **.rwdi** section stores the data for all the compressed data sections in the COFF file. Section 6.5.7 describes the **.rwdi** section.

6.5.2.2 Uninitialized Data Sections

To save space, the object file does not store zeros for uninitialized C global data. Instead, it groups such data into a single section named **.bss**. The **.bss** section has a section header but no section contents. The section header specifies the amount of uninitialized C global data in the program.

The linker determines which globals will be represented by the **.bss** section. The linker uses the symbol table to make this determination: after all linking is complete, any symbol whose storage class is external, section number is zero, and size is not zero is an uninitialized global and will be represented in **.bss**. The linker combines all input symbols of the same name into one symbol, whose size is equal to the size of the largest of its constituent symbols. The linker stores the total size of the **.bss** section in the **.bss** section header.

6.5.2.3 Loading Data Sections

The loader uses information from the section headers to allocate space for all data sections, then it writes data into the allocated space. If the data section has section contents, the loader copies those contents from the COFF file. If the section is an uninitialized or compressed data section (as identified by the section header's **STYP_BSS** flag), or if the section header's **STYP_ZERO** flag is set, the loader fills the allocated space with zeros. The loader follows the instructions in **.rwdi**, which describe how to write data into the space allocated for compressed data sections.

Before the program begins executing, the loader uses relocation information to modify the section contents. While the program is executing, information in the data section changes as data values change.

6.5.3 The .aptv Section

If a program uses absolute code, the loader cannot reposition the program's COFF file sections. The loader, therefore, does not need to change any of the virtual address references in the code. This allows the compiler to store the program's virtual address references with its machine instructions, in text sections. When address references are stored with machine instructions, instead of in a separate section, code is much more efficient.

Apollo COFF files may contain procedure calls to installed libraries, however, which are unresolved until load time. Rather than lose efficiency by storing all virtual address references in a separate read/write section, the COFF file may contain a read/write section for only those virtual addresses that are unknown at compile time.

The **Apollo transfer vector section**, named **.aptv**, is a data section found in Apollo COFF files that use absolute code and make calls to installed libraries. It is a read/write list of jump instructions to virtual addresses that are unknown at compile time.

If a procedure symbol is unresolved at compile time, the compiler can't write a call instruction to the procedure; instead, it writes a call instruction to a jump instruction, in the **.aptv** section. At compile time, the addresses in the **.aptv** section are all zero. At link time, the linker consolidates any duplicate jump instructions. Each jump instruction has a relocation entry, so the loader can write the correct addresses into the **.aptv** section's jump instructions at load time. At run time, control passes from the calling procedure to the **.aptv** section to the called procedure.

Table 6-9 shows the format of an entry in the **.aptv** section.

Table 6-9. Format of an Entry in the **.aptv** *Section*

Byte Offset	**Value**
0-1	JMP (the instruction to jump)
2-5	The virtual address to jump to

6.5.4 The .sri Section

Some Apollo COFF files contain a section called the **static resource information** section, or **.sri**. It contains information about the hardware and software resources that the program requires for execution.

The **.sri** section is composed of a header followed by any or all of the following resource records: a **hardware record**, a **software record**, a **systype record**, a **runtype record**, and a **stacksize record**. Each resource record lists the program's resource requirements for a particular type of resource.

The **coffdump** or **dump** command with the **-As** option displays an interpretation of the **.sri** records.

6.5.4.1 The .sri Header

The header is a 4-byte count of the number of resource records in the section.

6.5.4.2 Hardware Resource Records

A hardware resource record lists the hardware that the program requires. If these requirements are not met, the loader will not load the program.

Table 6–10 shows the format of a hardware resource record, and Table 6–11 describes the flags used in these records.

Table 6–10. Format of a Hardware Resource Record

Byte Offset	Field Name	Value
0–1	**kind**	The enumerated constant **hardware_sri_kind** (decimal value 1) identifying the record as a hardware resource record.
2–3	**combine**	An enumerated constant describing how the linker will combine this resource record with hardware resource records from other files. Table 6–15 lists the available values for **combine**.
4–7	**value**	A set of 1-bit flags identifying the hardware resources required. Table 6–11 describes each flag. **value** is the logical OR of all the applicable flags.

Table 6–11. Flags in a Hardware Resource Record

Hex Value	Flag Name	Meaning
0x01	**PEB_HARDWARE_FLAG**	The program requires a performance enhancement board.
0x02	**DNX60_HARDWARE_FLAG**	The program uses DNx60 instructions.
0x04	**M020_HARDWARE_FLAG**	The program uses 68020 instructions.
0x00008000	**MFPA1_HARDWARE_FLAG**	The program requires an FPA floating-point board.
0x00010000	**MFPX_HARDWARE_FLAG**	The program requires an FPX floating-point board.
0x00020000	**M881_HARDWARE_FLAG**	The program requries a 68881 coprocessor.

6.5.4.3 Software Resource Records

A **software resource record** lists any software requirements that the program imposes. The loader satisfies these requirements before executing the program. Table 6–12 shows the format of a software resource record.

Table 6–12. Format of a Software Resource Record

Byte Offset	Field Name	Value
0–1	**kind**	The enumerated constant **software_sri_kind** (decimal value 2) identifying the record as a software resource record.
2–3	**combine**	An enumerated constant describing how the linker will combine this resource record with software resource records from other files. Table 6–15 lists the available values for **combine**.
4–7	**value**	A set of 1-bit flags identifying the software resources required. Currently, the only software resource flag is **loadhigh_software_flag** (hex value 0x04). If this flag is set, the loader will load the program high in address space, leaving room to load other programs beneath it.

6.5.4.4 Systype and Runtype Resource Records

A **systype resource record** specifies the SYSTYPE environment variable (used, for example, to resolve environment-specific pathnames). A **runtype resource record** specifies the environment whose system call semantics the program requires. The system satisfies these requirements at load time.

Table 6-13 shows the format of a systype or a runtype resource record.

Table 6-13. Format of a Systype or Runtype Resource Record

Byte Offset	Field Name	Value
0-1	**kind**	The enumerated constant **systype_sri_kind** (decimal value 3) identifying the record as a systype resource record, or the enumerated constant **runtype_sri_kind** (decimal value 5) identifying the record as a runtype resource record.
2-3	**combine**	An enumerated constant describing how the linker will combine this resource record with systype or runtype resource records from other files. Table 6-15 lists the available values for **combine**.
4-5	**val_ms16**	One of the following decimal values, identifying the UNIX environment that the program requires. 0 SysV 1 BSD 2 Either SysV or BSD
6-7	**val_ls16**	One of the following decimal values, identifying the Apollo software release that the program requires. 0 SR8 1 SR9 2 SR10

6.5.4.5 Stacksize Resource Records

A **stacksize resource record** indicates the amount of stack space that the program requires. At run time, the loader allocates the requested space.

Table 6–14 shows the format of a stacksize resource record.

Table 6–14. Format of a Stacksize Resource Record

Byte Offset	Field Name	Value
0–1	**kind**	The enumerated constant **stacksize_sri_kind** (decimal value 6) identifying the record as a stacksize resource record.
2–3	**combine**	An enumerated constant describing how the linker will combine this resource record with stacksize resource records from other files. Table 6–15 lists the available values for **combine**.
4–7	**value**	The minimum required stack size, in bytes.

6.5.4.6 Combining Resource Records During Linking

When a linker combines files, it follows the instructions in the **combine** fields of each file's resource records to combine resource records of a single type. Hardware records, for example, typically instruct the linker to OR all hardware requirements. In a link operation, all resource records of a single type must specify the same rule for combining records; if not, the linker will not link the files.

Table 6–15 lists the rules that a linker may use to combine the requirements in resource records.

Table 6-15. Combining Rules

Decimal Value	Constant Name	Meaning
0	**TAKE_ALL_RULE**	Do not combine the requirements: include all **.sri** records in the output file.
1	**TAKE_SUM_RULE**	Sum the requirements.
2	**TAKE_MAX_RULE**	Use the largest requirement.
3	**TAKE_MIN_RULE**	Use the smallest requirement.
4	**TAKE_OR_RULE**	Perform logical OR with the requirements.
5	**TAKE_FIRST_RULE**	Use the requirement of the first file in the link operation.
6	**TAKE_LAST_RULE**	Use the requirement of the last file in the link operation.
7	**TAKE_UNIQUE_RULE**	Do not combine the requirements: include all unique **.sri** records in the output file. (Issue a warning and discard any duplicate **.sri** records.)
8	**TAKE_SPECIAL_RULE**	Use the combining rule specific to this type of **.sri** record. Runtype and systype records, for example, have "special" combining rules encoded into **bind** and **ld**.

6.5.5 The .mir Section

The module information section, named **.mir**, is an optional section found in some Apollo COFF files.

The **coffdump** or **dump** command with the **-Am** option displays an interpretation of the **.mir** records.

Domain compilers create one **.mir** section for each compilation unit. The **.mir** section created by the compiler contains one **name record** and one or more **maker records**. The name record points to the name of the module. The maker records identify all the tools used to create the module.

The linker preserves only the name record for the first module linked, and each unique maker record.

Table 6-16 shows the format of a name record.

Table 6–16. Format of a Name Record

Byte Offset	Field Name	Value
0–1	**kind**	The enumerated constant **module_name_mir_type** (decimal value 2) identifying the record as a name record.
2–3	**size**	8 (the size of the record, in bytes).
4–7	**module_name_offset**	Byte offset from the beginning of the COFF string table to the module name.

Each maker record describes one of the tools used to create the compilation unit. From a maker record, you can find

- The size of the maker record.
- The version of the software tool described by the maker record.
- When the software tool was created.
- The name of the tool.

Table 6–17 shows the format of a maker record.

Table 6–17. Format of a Maker Record

Byte Offset	Field Name	Value
0–1	**kind**	The enumerated constant **maker_mir_type** (decimal value 1) identifying the record as a maker record.
2–3	**size**	14 (the size of the record, in bytes).
4–5	**version**	The version number of the software tool described by this record.
6–9	**creation_time**	The number of seconds from 0:00:00 GMT 1/1/1970 to the time when the tool was created.
10–13	**maker_name_offset**	Byte offset from the beginning of the COFF string table to the tool's name.

6.5.6 The .inlib Section

An Apollo COFF file that references symbols in one or more shared libraries will have a shared library section, named **.inlib**. Chapter 4 describes shared libraries.

The **coffdump** or **dump** command with the **-Ai** option displays an interpretation of the **.inlib** section.

The first four bytes of the **.inlib** section contain the number of records in the section. The rest of the section is composed of **.inlib records**, one for each shared library that the program uses. Table 6-18 shows the format of an **.inlib** record.

Table 6-18. Format of an **.inlib** *Record*

Byte Offset	Field Name	Value
0-3	**reserved1**	0
4-7	**reserved2**	0
8-11	**reserved3**	0
12-15	**path_offset**	Byte offset from the beginning of the COFF string table to the shared library filename.

6.5.7 The .rwdi Section

The read/write data intialization section, named **.rwdi**, is an optional section found in some Apollo COFF files.

As we mentioned in Section 6.5.2.1, Domain compiler options determine whether data sections are stored in compressed form. A compressed data section has a section header that specifies the amount of address space that the loader must allocate for the section. The section has no contents. Instead, the section's data are represented in **.rwdi**.

The **coffdump** or **dump** command with the **-Ar** option displays an interpretation of the **.rwdi** records. With the **-Ar** option, **coffdump** or **dump** also displays the text of the **.rwdi** section in hexadecimal.

6.5.7.1 Records in the .rwdi Section

The **.rwdi** section is composed of text records, repeat records, and pad records.

A **text record** contains a block of data and the virtual address at which to load the data. A **repeat record** contains the number of consecutive times the data will be loaded into

the section. Not all text records are followed by repeat records; if a text record stands alone, its block of data is loaded once.

The loader will ignore any text record marked as a dead text record. This is useful if a tool processes the **.rwdi** section before load time. Rather than delete the processed records, the tool can simply mark them as dead.

For example, **bind** (the Aegis linker) includes an option to change data sections into text sections. Recall that text sections are read-only, and that the loader maps them directly from the COFF file into memory. At load time, therefore, the loader should not attempt to write to the converted section. If the data section is stored in uncompressed form, the linker simply changes the section header flag from **STYP_DATA** to **STYP_TEXT**. If the data section is stored in compressed form, the linker expands it by processing any **.rwdi** records for the section, then flags the section as a text section. Records in **.rwdi**, however, still instruct the loader to write over the section at load time. To negate those instructions, the linker marks as dead all the **.rwdi** text records that it processed. At load time, the loader ignores all dead text records and any repeat records following them.

Table 6-19 and Table 6-20 show the formats of text records and repeat records.

Table 6-19. Format of an **.rwdi** *Text Record*

Byte Offset	Field Name	Value
0-1	**kind**	The enumerated constant **rw_data_text** (decimal value 0) identifying the record as a text record, or the enumerated constant **rw_data_dead_text** (decimal value 2) identifying the record as a dead text record.
2-5	**target_vaddr**	The virtual address at which to begin loading the block of data.
6-9	**text_size**	The size of the block of data, in bytes.
10-end	**text**	An array containing the data to be loaded.

Table 6-20. Format of an **.rwdi** *Repeat Record*

Byte Offset	Field Name	Value
0-1	**kind**	The enumerated constant **rw_data_repeat** (decimal value 1) identifying the record as a repeat record.
2-5	**repeat_count**	The number of consecutive times to load the block of data contained in the preceding text record.

A **pad record** is simply two bytes of padding, useful for alignment purposes.

Table 6-21 shows the format of a pad record.

Table 6-21. Format of an **.rwdi** *Pad Record*

Byte Offset	Field Name	Value
0-1	**kind**	The enumerated constant **rw_data_pad** (decimal value 3) identifying the record as a pad record.

6.5.7.2 Using the .rwdi Section

The **.rwdi** section is useful for repetitive data. Representing a 100-element array of ones, for example, requires only two **.rwdi** records. The first is a text record containing 1 (the block of data to be loaded); the second is a repeat record containing 100 (the number of times to repeat the previous block of data).

The **.rwdi** section is also useful for representing a block of data that is largely uninitialized. Representing 10 bytes of data scattered among 1000 bytes of otherwise uninitialized data space requires, at most, only ten **.rwdi** text records. If any of those initialized bytes are consecutive, they can be represented together in a single **.rwdi** text record.

6.5.8 The .unwind Section

The **.unwind** section is a text section found in all Apollo COFF files. The section contains information for **unwinding** the stack, that is, restoring the stack and registers to the state they were in before the current block was called. (Typically, the block is a procedure or function.)

Tools such as Domain/DDE and DPAT rely on information in **.unwind**. On Series 10000 workstations, the information in **.unwind** is necessary to produce a traceback or to unwind the stack. On a 680x0-based workstation, when a function or procedure returns normally, it uses its own internal information to unwind the stack. When a function exits abnormally (for example, with a non-local goto or a cleanup handler), the operating system's stack unwinder uses information in the **.unwind** section to unwind the stack.

The **.unwind** section is composed of a series of **unwind descriptors**. From the information in a block's unwind descriptor, you can recreate the state of the stack and registers before the block began executing.

Every instruction address is described in an unwind descriptor, unless all the unwind information about that block is available without an unwind descriptor. An instruction address need not not appear in an unwind descriptor if it meets all three of these conditions:

1. The block containing the instruction doesn't push a stack frame,
2. The block does not change the values of any preserved registers, and
3. The block's return PC is in the standard location (at 4(SB) for a 680x0-based processor; in register .22 for a Series 10000 processor).

The **.unwind** section may contain **skip descriptors**, which serve as guides to the information in the section. It may also contain different kinds of unwind descriptors, depending on the processor type on which the program will run. Each of these descriptors is described in the following sections.

6.5.8.1 Skip Descriptors

Typically, the **.unwind** section includes several skip descriptors. Skip descriptors serve as headers placed periodically throughout the **.unwind** section and do not themselves contain unwind information. A skip descriptor tells you the range of PC values described by the group of descriptors immediately following it, and the number of bytes to the next group of descriptors.

Skip descriptors allow you to find the unwind descriptor for a specific PC range quickly. If the PC range in a skip descriptor doesn't include the range you're interested in, you can ignore the descriptors in its group and go directly to the next group of descriptors.

Table 6-22 shows the format of a skip descriptor.

Table 6-22. Format of a Skip Descriptor

Byte Offset	Field Name	Value
0-3	**start_pc**	The virtual address of the first PC described by this group of descriptors.
4-6	**length**	The range of PCs described by this group of descriptors. A program that will run on a Series 10000 workstation expresses **length** in bytes divided by 2; for a 680x0-based workstation, **length** is in bytes.
7	**kind**	The enumerated constant **udk$skip** (decimal value 4), identifying the record as a skip descriptor.
8-11	**skip_displacement**	The number of bytes from the beginning of the skip descriptor to the next group of descriptors.

If the PC range you're looking for does not lie between **start_pc** and **start_pc + length** (inclusive), read the next descriptor which is **skip_displacement** bytes from the beginning of the skip descriptor.

6.5.8.2 Series 10000 Descriptors

If a program will run on a Series 10000 workstation, its **.unwind** section uses Series 10000 unwind descriptors. The **.unwind** section uses compressed unwind descriptors to describe blocks that follow the standard Series 10000 linkage and register saving conventions; it describes other blocks by full unwind descriptors. In order to understand these descriptors, you must understand the Series 10000 calling conventions, described in Chapter 5.

From a Series 10000 compressed unwind descriptor, you can find

- The start of the block's instructions.
- The end of the block's instructions.
- Which of the integer (IP) registers from register **.10** to register **.23** are saved.
- Whether the block creates a stack frame.
- The stack pointer.
- The locations of the saved register values.
- Which of the floating-point (FP) registers from register **.FS4** to **.FS7** are saved.
- Which register is saved at [SF,4].
- The return PC.
- Whether the entry control block pushed the stack frame.
- Where in the execution of the machine instructions the register values are saved.
- Where in the execution of the machine instructions the stack frame is pushed.

From a Series 10000 full unwind descriptor, you can find all the information stored in a Series 10000 compressed unwind descriptor, as well as the following

- Which of the IP registers from register **.0** to register **.31** are saved.
- Which of the FP registers from **.FS1** to **.FS63** are saved.

Every unwind descriptor for the Series 10000 processor contains the fields described in Table 6–23. A full unwind descriptor also contains the additional fields described in Table 6–24.

Table 6-23. Fields Contained in Every Series 10000 Unwind Descriptor

Byte:Bit Offset	Field Name	Value
0–3	**start_pc**	The virtual address of the first instruction for this block.
4–6	**length**	The length of the block's instructions, in words (bytes divided by 2).
7	**kind**	The enumerated constant **udk$a88k_compressed** (decimal value 0) if the record is a compressed unwind descriptor, or the enumerated constant **udk$a88k_full** (decimal value 2) if the record is a full unwind descriptor.
8:7–9:2	**mask_saved_ip_regs_sml**	14 bits identifying which of the IP registers from .10 to .23 are saved. The bits correspond to those IP registers, in order: the low bit represents IP register .10; the high bit represents IP register .23. If a bit is set, its corresponding register is saved.
9:1	**is_sf_pushed**	A Boolean variable, true if the block creates a stack frame.
9:0	**is_os_stack**	A Boolean variable, true if the stack pointer is register .26; false if the stack pointer is register .23.
10:7–11:4	**sf_offset_regs_saved**	The offset from the stack pointer, in longwords (bytes divided by 4), to the saved register values. Register values are always saved in the following order: all saved IP registers, followed by all saved FP registers, in ascending order by register number.
11:3–11:0	**mask_saved_fp_regs_sml**	4 bits identifying which of the FP registers from FS4 to FS7 are saved. The bits correspond to those FP registers, in order: the low bit represents register FS4; the high bit represents register FS7. If a bit is set, its corresponding register is saved.
12:7–12:3	**register_saved_at_4_sf**	The number of the IP register saved at [.sf,4], or 31 if no register is saved at [.sf,4].
12:2–13:6	**register_saved_at_12_sf**	The number of the IP register saved at [.sf,12], or 31 if no register is saved at [.sf,12]

(Continued)

Table 6-23. Fields Contained in Every Series 10000 Unwind Descriptor (Cont.)

Byte:Bit Offset	Field Name	Value
13:5–13:1	**register_return**	The number of the IP register where the return PC is stored (usually 22).
13:0	**is_return_saved**	A Boolean variable, true if the return PC is on the stack (at [.sf,8]).
14:7	**is_sf_pushed_in_ecb**	A Boolean variable, true if the entry control block pushed the stack frame.
14:6–15:7	**pc_offset_regs_saved**	The PC offset into the block at which register values are saved. Before this offset, the block has not modified the values of the registers that will be saved. When the PC is at this offset, the block has finished saving register values.
15:6–15:3	**pc_offset_sf_pushed**	A PC offset into the block. When the PC is at this offset, the stack frame has been pushed. This field is ignored if **is_sf_pushed_in_ecb** is true.

Table 6-24. Additional Fields Contained in a Series 10000 Full Unwind Descriptor

Byte:Bit Offset	Field Name	Value
15:2–18:2	**mask_saved_ip_registers**	32 bits identifying which IP registers are saved. The bits correspond to the IP registers, in order: the low bit represents IP register .0; the high bit represents IP register .31. If a bit is set, its corresponding register is saved. When this field is used, **mask_saved_ip_regs_sml** is ignored.
18:1–21:1	**mask_saved_fp_registers_high**	32 bits identifying which high FP registers are saved. The **_registers_high** bits correspond to the high FP registers, in order: the low bit represents register FS32; the high bit represents register FS63. If a bit is set, its corresponding register is saved. When this field is used, **mask_saved_fp_regs_sml** is ignored.
21:0–24:0	**mask_saved_fp_registers_low**	32 bits identifying which low FP registers are saved. The bits correspond to the low FP registers, in order: the low bit represents register FS0; the high bit represents register FS31. If a bit is set, its corresponding register is saved. When this field is used, **mask_saved_fp_regs_sml** is ignored.

6.5.8.3 MC68000 Descriptors

If a program will run on an MC68000-based workstation, its **.unwind** section uses MC68000 unwind descriptors. Blocks that follow the standard MC68000 linkage and register saving conventions are described by compressed unwind descriptors; other blocks are described by full unwind descriptors. In order to understand these descriptors, you must understand the MC68000 calling conventions, described in Chapter 5.

From an MC68000 compressed unwind descriptor, you can find

- The start of the block's instructions
- The end of the block's instructions
- Which of the address registers are saved
- Which of the data registers are saved
- Which of the floating point registers are saved
- The locations of the saved register values

From an MC68000 full unwind descriptor, you can find all the information stored in an MC68000 compressed unwind descriptor, as well as the following:

- Whether offsets in the descriptor are relative to the stack base or to the stack pointer
- Whether the block requires an unwind descriptor
- Whether the block uses completely nonstandard conventions
- The location of the return PC, if it isn't 4(SB)

Every unwind descriptor for the MC68000 processor contains the fields described in Table 6-25. A full unwind descriptor contains the additional fields described in Table 6-26, as well.

Table 6-25. Fields Contained in Every MC68000 Unwind Descriptor

Byte Offset	Field Name	Value
0-3	**start_pc**	The virtual address of the first instruction for this block.
4-6	**length**	The length of the block's instructions, in bytes div 2.
7	**kind**	The enumerated constant **udk$m68k_compressed** (decimal value 1) if the record is a compressed unwind descriptor, or the enumerated constant **udk$m68k_full** (decimal value 3) if the record is a full unwind descriptor.
8	**saved_a_regs**	8 bits identifying which of the address registers are saved. The bits correspond to the address registers, in order: the low bit represents register a0; the high bit represents register a7. If a bit is set, its corresponding register is saved.
9	**saved_d_regs**	8 bits identifying which of the data registers are saved. The bits correspond to the data registers, in order: the low bit represents register d0; the high bit represents register d7. If a bit is set, its corresponding register is saved.
10-11	**saved_fp_regs**	16 bits identifying which of the floating-point registers are saved. The bits correspond to the floating-point registers, in order: the low bit represents register fp0; the high bit represents register fp15. If a bit is set, its corresponding register is saved.
12-15	**saved_regs_off**	The offset from the stack base, in bytes, to the saved register values. Register values are always saved in the following order: all saved floating-point registers, followed by all saved data registers, followed by all saved address registers, in ascending order by register number.

Table 6-26. Additional Fields Contained in MC68000 Full Unwind Descriptors

Byte:Bit Offset	Field Name	Value
16:7	**sp_rel**	A Boolean value, true if the current routine does not set the stack base register, but specifies the saved register location and the return PC location as offsets from the stack pointer. When **sp_rel** is true, **pc_offset** (described below) is required. If the current routine sets the stack base register, then **sp_rel** is false, offsets are relative to the stack base, the return PC is at 4(SB), and **pc_offset** is ignored.
16:6	**no_stack**	A Boolean value, true if the block changes no registers and pushes no data onto the stack. If **no_stack** is true, this block does not require an unwind descriptor; the unwind descriptor appears for completeness only.
16:5	**non_std**	A Boolean value, true if the block uses completely non-standard conventions. The register masks (**saved_a_regs**, **saved_d_regs**, and **saved_fp_regs**) indicate which registers are modified, but the register values are unrecoverable. The return PC is also unknown. Consequently, a traceback generally fails if it encounters non-standard code.
20-23	**pc_offset**	Offset, from the stack pointer, to the return PC. This value is used only if **sp_rel** is true. If **sp_rel** is false, the return PC is at 4(SB) and **pc_offset** is ignored.

6.5.9 The .blocks Section

The **.blocks** section is one of the three additional COFF sections that Domain compilers and assemblers create to support high-level language debugging. Both **dbx** and the Domain Distributed Debugging Environment (Domain/DDE) use these sections.

The **.blocks** section is the starting point for finding all of the information (outside of the **.unwind** section) that a debugger needs. The **.blocks** section does the following:

- Organizes the program into lexical blocks of source code (such as modules, compilation units, procedures, functions, and subroutines). If we consider a program's hierarchical block structure, each block has a **parent block** (the block in which it's defined). It may have one or more **sibling blocks** (other blocks defined in the same parent block). In some languages, it may have one or more **child blocks** (blocks defined in it). In the **.blocks** section, each block points to its next sibling block and its first child block.
- Maps each block to its corresponding machine instructions.
- Points to the other debug information for each block.

The compilers generate a single **.blocks** section for each compilation unit. The **.blocks** section for one compilation unit contains a **header**, a **section table**, a **file table**, one or more **string tables**, and a series of **block records**. The **.blocks** section may also include one or more **auxiliary records**, **pad records**, and **extension records**, described at the end of this section.

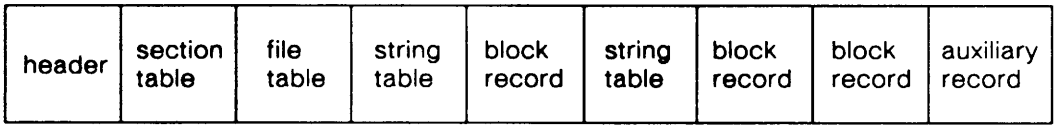


Figure 6–2. The Structure of a Sample **.blocks** *Section for a Single Compilation Unit*

Since the linker concatenates the **.blocks** sections, the **.blocks** section in a COFF file may be composed of consecutive **.blocks units**, each of which represents one compilation unit. Each **.blocks** section starts with a header record.

6.5.9.1 Header

The header record describes the compilation unit as a whole. From the header record, you can find

- The version of the header file in which the **.blocks** records are defined.
- The source language in which the compilation unit was written.
- The number of blocks in the compilation unit.
- The block records for the compilation unit.
- The section table for the compilation unit.
- The file table for the compilation unit.
- The size of the **.blocks** unit for the compilation unit.

Table 6–27 shows the format of the header record, and Table 6–28 lists the defined constants which identify the source language in which the compilation unit was written.

Table 6-27. Format of a **.blocks** *Header Record*

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_comp_unit** (decimal value 1), identifying the record as a header record.
1	**rec_flags**	0
2	**version.major_part**	The major part of the version number of the debug header file (**dst.h**) used in this COFF file. For example, if this COFF file's debug sections are defined in version 1.3 of **dst.h**, **version.major_part** is 1.
3	**version.minor_part**	The minor part of the version number of the debug header file (**dst.h**) used in this COFF file. For example, if this COFF file's debug sections are defined in version 1.3 of **dst.h**, **version.minor_part** is 3.
4–5	**flags**	A flag indicating how the **.symbols** section specifies ranges in location strings. If bit 0 (the low bit) is set, **.symbols** specifies ranges in number of PC locations. If bit 0 is not set, **.symbols** specifies ranges in number of lines of source code. Section 6.5.10.13 describes location strings and ranges.
6–7	**lang_type**	An enumerated constant identifying the source language in which the compilation unit was written. Table 6–28 lists the available values.
8–11	**number_of_blocks**	The number of blocks in the compilation unit.
12–15	**root_block_offset**	Byte offset from the beginning of the header record to the first block record for the compilation unit.
16–19	**section_table**	Byte offset from the beginning of the header record to the **.blocks** section table for the compilation unit.
20–23	**file_table**	Byte offset from the beginning of the header record to the file table for the compilation unit.
24–27	**data_size**	The size of the **.blocks** unit for the compilation unit, including padding, in bytes.

Table 6–28. Enumerated Constants Identifying the Source Language

Decimal Value	Constant Name	Language
0	**DST_LANG_UNK**	Unknown
1	**DST_LANG_PAS**	Pascal
2	**DST_LANG_FTN**	FORTRAN
3	**DST_LANG_C**	C
4	**DST_LANG_MOD2**	Modula–2
5	**DST_LANG_ASM_M68K**	680x0 assembly language
6	**DST_LANG_ASM_A88K**	Series 10000 assembly language
7	**DST_LANG_ADA**	Ada
8	**DST_LANG_CXX**	C++

6.5.9.2 Section Table

When the linker combines COFF files, it concatenates or overlays the sections from the component files. Consider, for example, linking **filea** and **fileb** to form **fileab**. The **.text** sections from **filea** and **fileb** are combined into a single **.text** section in **fileab**. The **.text** section header provides the virtual starting addresses of the new, single **.text** section. It provides no information, however, on which part of **.text** came from which component file.

The **.blocks** section table pertains to a single compilation unit, however. A **.blocks** section table entry, therefore, points to a single compilation unit's contribution to a section. The **filea .blocks** section table, for example, lists the virtual address of **filea**'s contribution to the **.text** section, as shown in Figure 6–3.

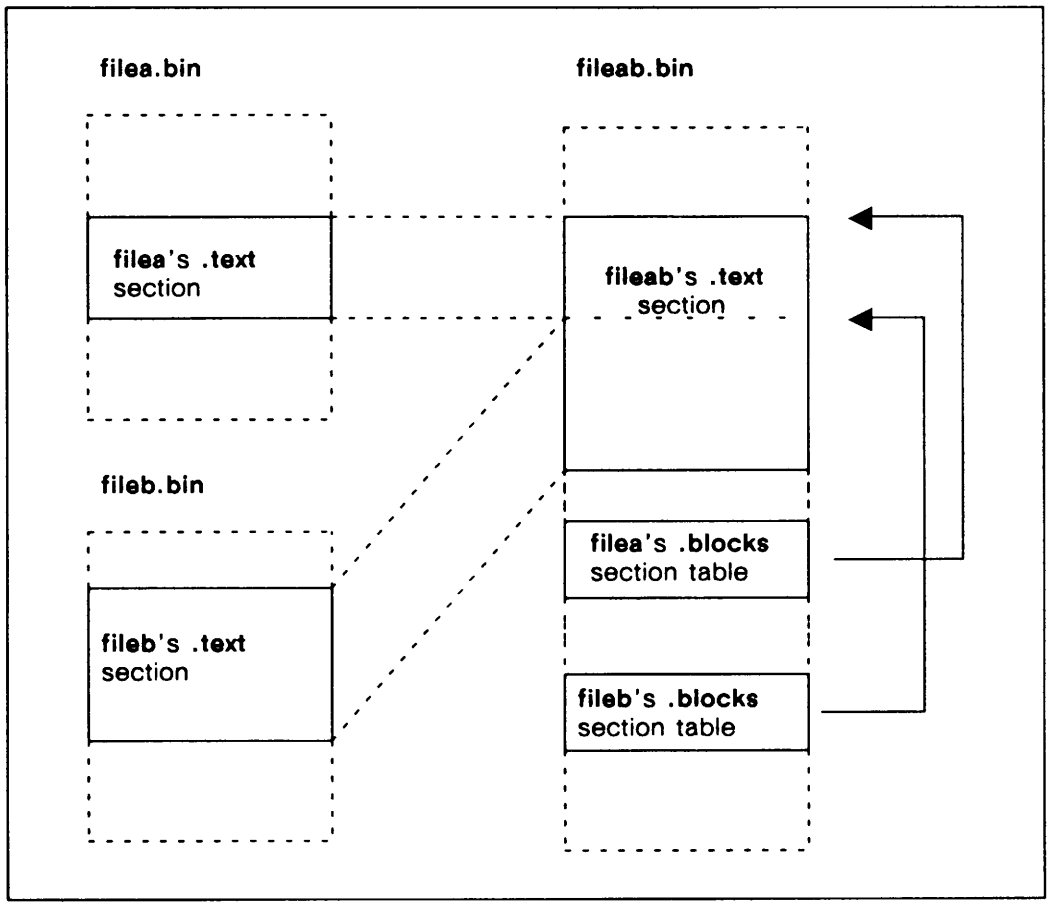


Figure 6–3. Two **.blocks** *Section Tables*

Table 6–29 shows the format of the **.blocks** section table.

Table 6–29. Format of a **.blocks** *Section Table*

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_section_tab** (decimal value 2), identifying the record as a section table.
1	**rec_flags**	0
2–3	**number_of_sections**	The number of addresses listed in the table.
4–end	**section_base**	An array of 4-byte virtual addresses.

The section table stores the starting virtual address of each of the sections pointed into by block records. It may store other virtual addresses, as well, but the purpose of the section table is to allow block records to use section-relative addresses. For example, instead of pointing directly to the machine instructions corresponding to a block of source code, the block record stores a section table index (origin at 1) and an offset. The index identifies the symbol table entry that points to the beginning of the **.text** section (or to another text section that holds the instructions). The offset locates the block's instructions within the section. Figure 6-4 illustrates how block records use the **.blocks** section table.

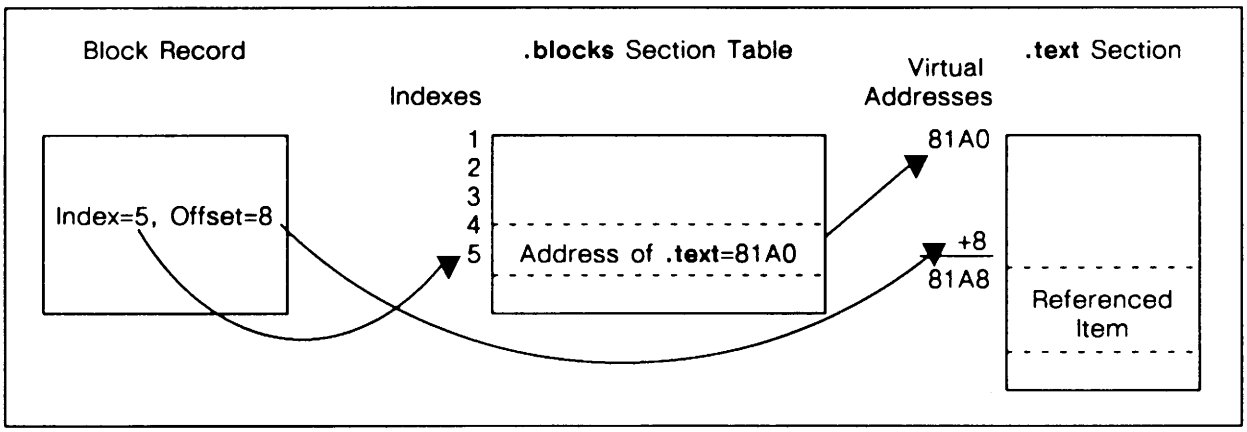


Figure 6-4. How Block Records Use the **.blocks** *Section Table*

6.5.9.3 File Table

The file table describes all of the source and include files that make up the compilation unit. From the file table, you can find

- The number of files described in the table
- The date and time when each file was last modified
- The name of each file

Table 6-30 shows the format of the file table.

NOTE: The Apollo system clock is 48 bits long, and counts 4-microsecond increments from 00:00:00 1/1/1980 (UTC). The **time_$clockh_t** format used in the **.blocks** file table contains only the most significant 32 bits of the clock.

Table 6-30. Format of a **.blocks** *File Table*

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_file_tab** (decimal value 3), identifying the record as a file table.
1	**rec_flags**	0
2–3	**number_of_files**	The number of files listed in the table.
4–end	**files**	An array of entries, one for each file. Each entry is composed of the following fields:

Byte Offset	Field Name	Value
0–3	**dtm**	The time in **time_$clockh_t** format when the file was last modified.
4–7	**noffset**	Byte offset from the begin ning of the file table to the filename.

6.5.9.4 String Table

The string tables contain the strings to which other records in the **.blocks** section refer. Table 6-31 shows the format of a string table.

Table 6-31. Format of a String Table

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_string_tab** (decimal value 27), identifying the record as a string table.
1	**rec_flags**	0
2–3	**length**	The length of the table, in bytes.
4–end	**text**	An array of null-terminated strings.

6.5.9.5 Block Records

Block records describe the block structure of the module. Each block record describes a single block. From a block record, you can find

- Whether the record is followed by any auxiliary records
- The block's type (for example, module, program, procedure, function, subroutine)
- The block record for the block's next sibling
- The block record for the block's first child
- The block's name
- The symbol information (in the **.symbols** section) for the block
- The number of separate sub-blocks into which the block is divided. (If a block's source code is all stored contiguously, and if its machine instructions are all stored contiguously, the block consists of one sub-block. Otherwise, each of the separate pieces of the block's code or instructions is a sub-block.)
- The size of each sub-block
- The machine instructions for each sub-block
- The line number information (in the **.lines** section) for each sub-block

Table 6-32 shows the format of a block record. Table 6-33 describes the flags which may appear in a block record. Table 6-34 lists the constants which identify a block's type, and Table 6-35 shows the format of an entry in a **code_ranges** array.

Table 6-32. Format of a Block Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_block** (decimal value 4), identifying the record as a block record.
1	**rec_flags**	A set of 8 bits. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2	**block_type**	An enumerated constant identifying the block's type, for descriptive purposes only. Table 6-34 lists the available values.
3	**flags**	A set of flags describing some features of the block. Table 6-33 describes each flag. **flags** is the logical OR of all the applicable flags.
4-7	**sibling_block_off**	Byte offset from the beginning of the block record to the block record for the next lexical sibling block (if any).
8-11	**child_block_off**	Byte offset from the beginning of the block record to the block record for the first lexical child block (if any).
12-15	**noffset**	Byte offset from the beginning of the block record to the block name.
16-17	**sect_index**	The section table index for the **.symbols** section.
18-21	**sect_offset**	Byte offset from the beginning of the **.symbols** section to the symbol information for this block.
22-23	**n_of_code_ranges**	The number of sub-blocks in the block; 1 if the block is not divided.
24-end	**code_ranges**	An array of entries, one for each sub-block. Table 6-35 shows the format of an entry in the **code_ranges** array.

Table 6–33. Flags in a Block Record

Hex Value	Flag Name	Meaning
0x01	**DST_BLOCK_MAIN_ENTRY**	This block is the main program. Domain/DDE stops in this block when starting the program.
0x02	**DST_BLOCK_EXECUTABLE**	This block has one or more entry points.

Table 6–34. Enumerated Constants Identifying a Block's Type

Decimal Value	Constant Name	Block Type
0	**DST_BLOCK_MODULE**	Module
1	**DST_BLOCK_PROGRAM**	Program
2	**DST_BLOCK_PROCEDURE**	Procedure
3	**DST_BLOCK_FUNCTION**	Function
4	**DST_BLOCK_SUBROUTINE**	Subroutine
5	**DST_BLOCK_BLOCK_DATA**	FORTRAN block data
6	**DST_BLOCK_STMT_FUNCTION**	FORTRAN statement function
7	**DST_BLOCK_PACKAGE**	Ada package
8	**DST_BLOCK_PACKAGE_BODY**	Ada package body
9	**DST_BLOCK_SUBUNIT**	Ada subunit
10	**DST_BLOCK_TASK**	Ada task
11	**DST_BLOCK_FILE**	File scope
12	**DST_BLOCK_CLASS**	C++ or Simula class block

Table 6–35. Format of an Entry in a **code_ranges** *Array*

Byte Offset	Field Name	Value
0–3	**code_size**	The size of the executable code in the sub-block, in bytes.
4–5	**code_start.sect_index**	The section table index for the section containing the sub-block's machine instructions.
6–9	**code_start.sect_offset**	Byte offset from the beginning of the section to this sub-block's machine instructions.
10–11	**lines_start.sect_index**	The section table index for the **.lines** section.
12–15	**lines_start.sect_offset**	Byte offset from the beginning of the **.lines** section to this sub-block's line number information.

6.5.9.6 Auxiliary Records

A block record may be modified by an auxiliary source range record. The auxiliary record, if used, directly follows the block record that it modifies.

An auxiliary source range record indicates a range of source lines to which the information in the block record applies. Table 6–36 shows the format of an auxiliary source range record.

Table 6–36. Format of an Auxiliary Source Range Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_src_range** (decimal value 51), identifying the record as an auxiliary source range record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records. If bit 1 (the second lowest bit) is set, the record is the last in a series of auxiliary records.
2–5	**first_line**	The source code line number, from the beginning of the source file, of the first line in the range.
6–9	**last_line**	The source code line number, from the beginning of the source file, of the last line in the range.

6.5.9.7 Pad Records

A pad record contains no data, but provides 2 bytes of padding, for alignment purposes. Table 6-37 shows the format of a pad record.

Table 6-37. Format of a Pad Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_pad** (decimal value 0), identifying the record as a pad record.
1	**rec_flags**	0

6.5.9.8 Extension Records

We provide extension records to allow compilers to generate records other than the ones defined in this section. Domain/DDE and **dbx** ignore these records. Table 6-38 shows the format of an extension record.

Table 6-38. Format of an Extension Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_extension** (decimal value 37), identifying the record as an extension record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records. If bit 1 (the second lowest bit) is set, the record is the last in a series of auxiliary records.
2-3	**rec_size**	The size of the record, including the **rec_type** and **rec_flags** fields, in bytes.
4-5	**ext_type**	A compiler-defined structure identifying the item's data type.
6-7	**ext_data**	A compiler-defined structure containing the item's data.

6.5.10 The .symbols Section

The **.symbols** section is another of the three debugging sections. It describes all of the symbols—named constants, variables, procedure names, function names, and program labels—in the program. It serves the same purpose as the COFF symbol table, but it provides more information.

The **.symbols** section is organized by blocks: each block defined in **.blocks** has its own portion of the **.symbols** section, describing all the symbols used in that block.

> NOTE: At SR10.2, we changed the format of the **.symbols** section to improve the efficiency of Domain/DDE and **dbx**. This section describes the new format. The header file **/usr/include/apollo/dst.h** defines the new format. If you are writing a compiler to generate code that Domain/DDE can debug, use this format.
>
> At this writing, Domain compilers, assemblers, and linkers still produce the **.symbols** format instituted at SR10, which is similar but not identical to the format described in this section.

6.5.10.1 Records in the .symbols Section

The **.symbols** section contains the following types of records:

- **Symbol records**, which describe individual symbols.
- **Type records**, which describe user-defined data types. Symbol records for user-defined symbols point to type records.
- **Auxiliary records**, which modify symbol records and type records.

For each symbol, the **.symbols** section contains a **symbol record**. From the symbol record, you can find the symbol's name, type, and the source code line on which it's defined. Section 6.5.10.3 describes the formats of symbol records.

Within a symbol record, the **type descriptor** field identifies the symbol's type. Section 6.5.10.4 describes the format of a type descriptor.

For symbols with standard types, the type descriptor includes an enumerated value representing the data type, as illustrated in Figure 6–5.

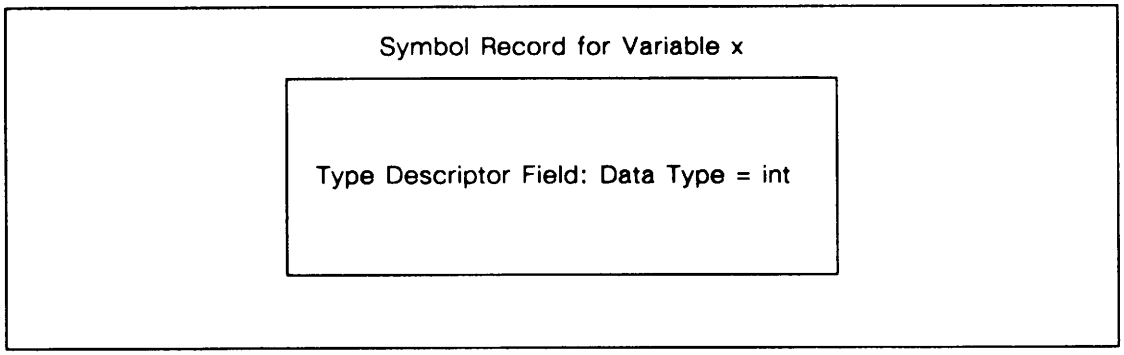


Figure 6-5. Type Descriptor Field Naming a Standard Data Type.

If a symbol's type is user-defined, the type descriptor field points to a **type record**, as illustrated in Figure 6-6.

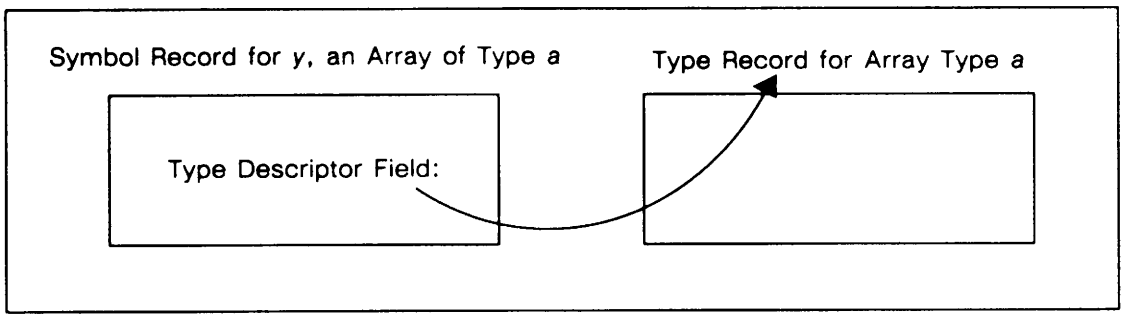


Figure 6-6. Type Descriptor Field Pointing to a User-Defined Type Record.

The type record defines the data type. From a type record, you can find the type's name, the source code line on which it's defined, the type(s) that compose it, and any other information in the type definition. Section 6.5.10.5 describes the formats of type records.

Several types of **auxiliary records** modify symbol records or type descriptors. For example, an auxliary size record modifies the default size of a type descriptor. Section 6.5.10.6 describes the formats of auxiliary records.

A few miscellaneous other records may appear in **.symbols**, as well. These records include string tables that hold null-terminated strings. Symbol records, type records, and auxiliary records will point into string tables to find name strings. These records include pad records that hold zeros and appear for alignment purposes, forward records that simply reference other records, and end scope records that divide the **.symbols** section into groups for the loader. Sections 6.5.10.8 through 6.5.10.11 describe the formats of these other records.

Figure 6-7 shows the structure of a sample **.symbols** section.

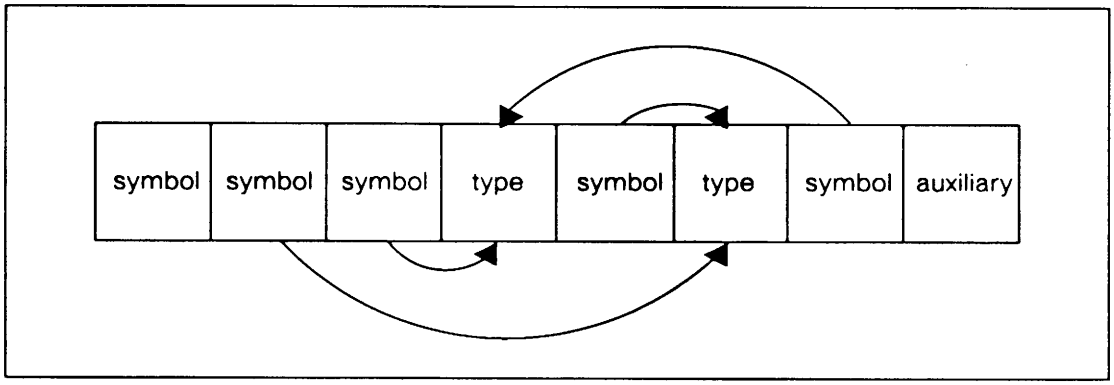


Figure 6–7. The Structure of a Sample **.symbols** *Section*

6.5.10.2 How .symbols Records Specify Locations

The records in the **.symbols** section use the following three structures to describe locations:

- **Offsets** are the number of bytes from one position in the object file to another.
- **Source code locations**, described in Section 6.5.10.12, identify locations within the source code.
- **Location strings**, described in Section 6.5.10.13, identify locations in address space.

6.5.10.3 Symbol Records

The **.symbols** section includes a symbol record for each symbol defined in a block. A symbol record is one of the following:

- Constant record
- Variable record
- Entry record
- Label record

Constant Records. A constant is a program element that has a value, but no location in memory. The **.symbols** section includes a constant record for each named constant defined in a program. From a constant record, you can find:

- The name of the constant
- Where in the source code the constant is defined

- The constant's data type
- The size of the constant
- The constant's value

Table 6-39 shows the format of a constant record.

Table 6-39. Format of a Constant Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated value **dst_typ_constant** (decimal value 40), identifying the record as a constant record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2-5	**noffset**	Byte offset from the beginning of the constant record to the constant's name string.
6-9	**src_loc**	The source code location where the constant is defined.
10-13	**type_desc**	A type descriptor describing the constant's data type.
14-15	**length**	The size of the constant, in bytes.
16-end	**val**	The constant's value.

Variable Records. The **.symbols** section includes a variable record for each variable used in a block. From a variable record you can find:

- The name of the variable
- The variable's value
- Where in the source code the variable is defined
- The variable's data type
- Whether the variable is read-only, volatile, global, compiler-generated, and/or static

Table 6-40 shows the format of a variable record, and Table 6-41 describes the flags used in a variable record.

Table 6-40. Format of a Variable Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated value **dst_typ_variable** (decimal value 46), identifying the record as a variable record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**noffset**	Byte offset, from the beginning of the variable record, to the variable's name string.
6–9	**loffset**	Byte offset, from the beginning of the variable, to the location string for the variable's value.
10–13	**src_loc**	The source code location where the variable is defined.
14–17	**type_desc**	A type descriptor describing the variable's data type.
18–19	**attributes**	A set of flags describing other attributes of the variable. Table 6–41 lists the available values. The value of **attributes** is the logical OR of the values of the applicable flags.

Table 6-41. Flags in a Variable Record

Hex Value	Flag Name	Meaning
0x01	**DST_VAR_ATTR_READ_ONLY**	The variable is read-only (a program literal).
0x02	**DST_VAR_ATTR_VOLATILE**	The variable is volatile.
0x04	**DST_VAR_ATTR_GLOBAL**	The variable is a global definition or reference.
0x08	**DST_VAR_ATTR_COMPILER_GEN**	The variable is compiler-generated.
0x10	**DST_VAR_ATTR_STATIC**	The variable has a static location.

Entry Records. The **.symbols** section includes an entry record for each function or procedure defined in or called from a block. From an entry record you can find

- The name of the routine.
- The routine's entry point (its first instruction).
- Where in the source code the routine is defined.
- The routine's signature (the description of its arguments and return value).
- For routines with multiple entry points, which entry point this record describes.

Table 6-42 shows the format of an entry record.

Table 6-42. Format of an Entry Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated value **dst_typ_entry** (decimal value 48), identifying the record as an entry record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**noffset**	Byte offset from the beginning of the entry record to the routine's name string.
6–9	**loffset**	Byte offset from the beginning of the entry record to the location string for the routine's entry point.
10–13	**src_loc**	The source code location where the routine is defined.
14–17	**sig_desc**	Byte offset from the beginning of the entry record to the signature type record for this routine.
18	**entry_number**	For routines with multiple entry points, the number of the entry point described by this entry record.

Label Records. A label is a location in memory that has no associated data type. The **.symbols** section includes a label record for each label used in a block. From a label record you can find

- The label's name
- The label's location

- Where in the source code the label is defined

Table 6-43 shows the format of a label record.

Table 6-43. Format of a Label Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated value **dst_typ_label** (decimal value 47), identifying the record as a label record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2-5	**noffset**	Byte offset from the beginning of the label record to the name string for the label.
6-9	**loffset**	Byte offset from the beginning of the label record to the location string for the label.
10-13	**src_loc**	The source code location where the label is defined.

6.5.10.4 Type Descriptors

A type descriptor takes one of two forms. If it describes a standard data type, a type descriptor names the data type, as shown in Table 6-44. If it describes a user-defined type, a type descriptor points to the appropriate type record, as shown in Table 6-46.

Table 6-44. Format of a Type Descriptor for a Standard Data Type

Byte:Bit Offset	Field Name	Value
3:0	**user_defined_type**	A Boolean false (0).
0	**dtc**	An enumerated constant identifying the data type. Table 6-45 lists the available values.

Table 6-45. Enumerated Constants Identifying a Standard Data Type

Decimal Value	Constant Name	Data Type
0	**DST_INT8_TYPE**	8-bit integer
1	**DST_INT16_TYPE**	16-bit integer
2	**DST_INT32_TYPE**	32-bit integer
3	**DST_UINT8_TYPE**	8-bit unsigned integer
4	**DST_UINT16_TYPE**	16-bit unsigned integer
5	**DST_UINT32_TYPE**	32-bit unsigned integer
6	**DST_REAL32_TYPE**	Single-precision IEEE floating-point value
7	**DST_REAL64_TYPE**	Double-precision IEEE floating-point value
8	**DST_COMPLEX_TYPE**	Single-precision complex value
9	**DST_DCOMPLEX_TYPE**	Double-precision complex value
10	**DST_BOOL8_TYPE**	8-bit Boolean value
11	**DST_BOOL16_TYPE**	16-bit Boolean value
12	**DST_BOOL32_TYPE**	32-bit Boolean value
13	**DST_CHAR_TYPE**	8-bit ASCII character
14	**DST_STRING_TYPE**	String of 8-bit ASCII characters
15	**DST_PTR_TYPE**	Pointer
16	**DST_SET_TYPE**	Set of 256 bits
17	**DST_PROC_TYPE**	Procedure (signature unspecified)
18	**DST_FUNC_TYPE**	Function (signature unspecified)
19	**DST_VOID_TYPE**	C void type value
20	**DST_UCHAR_TYPE**	C unsigned char type value

Table 6–46. Format of a Type Descriptor for a User–Defined Type

Byte:Bit Offset	Field Name	Value
3:0	**user_defined_type**	A Boolean true (1).
0–3:1	**doffset**	Byte offset from the beginning of the record containing the type descriptor to the type record that defines the user–defined type.

6.5.10.5 Type Records

Type records describe user–defined data types. Entry records point to the type record for the routine's signature. The type descriptor field of a symbol record for a user–defined type points to a type record. Type records are available for any of the following data types:

- Pointer
- Array
- Subrange
- String
- Set
- Implicit enumeration
- Explicit enumeration
- Record/union
- File
- Alias
- Signature

Pointer Records. A pointer record describes a pointer or reference type. From a pointer record, you can find

- The name of the pointer or reference type
- Where in the source code the pointer or reference type is defined
- The data type of the items to which the pointer or reference points

Table 6–47 shows the format of a pointer record.

Table 6-47. Format of a Pointer Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated value **dst_typ_pointer** (decimal value 7), identifying the record as an pointer record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2-5	**noffset**	Byte offset from the beginning of the pointer record to the pointer or reference type's name string.
6-9	**src_loc**	The source code location where the pointer or reference type is defined.
10-13	**type_desc**	A type descriptor describing the data type to which the pointer or reference points. Type descriptors, usually found in symbol records, are described in Section 6.5.10.4.

Array Records. An array record describes an array type. From an array record, you can find

- Whether any auxiliary records modify the record.
- The array type's name.
- Where in the source code the array type is defined.
- The data type of the array elements.
- The data type of the array indexes.
- The lower bound of the array.
- The upper bound of the array.
- The size of the array.
- Whether the array is packed.
- Whether packed array elements are signed.
- The size of packed elements.
- Whether the array is part of a multi-dimensional array.

- The distance between successive elements in the array. (This information is useful for multi-dimensional arrays.)
- The order in which array dimensions are arranged in memory.

Table 6-48 shows the format of an array record.

Table 6-48. Format of an Array Record

Byte:Bit Offset	Field Name	Value
0	**rec_type**	The enumerated value **dst_typ_array** (decimal value 8), identifying the record as an array record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**noffset**	Byte offset from the beginning of the array record to the array type's name string.
6–9	**src_loc**	The source code location where the array type is defined.
10–13	**elem_type_desc**	A type descriptor describing the data type of the array elements.
14–17	**indx_type_desc**	A type descriptor describing the data type of the array indexes.
18–21	**lo_bound**	The lowest array index value.
22–25	**hi_bound**	The highest array index value.
26–29	**span**	If the array has constant bounds, this value is the number of bytes between successive elements in the array. If the array has variable bounds, this value is the alignment padding, in bytes, and **span_comp** describes how to calculate the distance between array elements.
30–33	**size**	The size of the array, in bytes.
34:7	**multi_dim**	A Boolean value, true if the array is part of a multi-dimensional array.
34:6	**is_packed**	A Boolean value, true if the array is packed.
34:5	**is_signed**	A Boolean value, true if the array is packed and the packed elements contain signed values.

(Continued)

Table 6-48. Format of an Array Record (Cont.)

Byte Offset	Field Name	Value
34:4-34:3	**span_comp**	If the array has constant bounds, this value is the enumerated constant **dst_use_span_field** (decimal value 0), indicating that the distance between elements is the value in **span.** If the array has variable bounds, this value is the enumerated constant **dst_compute_from_next** (decimal value 1). The distance between elements is the size of the elements in the next array dimension, plus any padding. The value in **span** is the padding.
34:2	**column_major**	A Boolean value, true for arrays stored in column-major form, such as in FORTRAN, where the first array bound in the declaration varies the most rapidly in memory. For example, a FORTRAN array declared as **A**[*m*,*n*] is stored as *n* groups of *m* items. (Languages that use row-major arrays, such as Pascal and C, store an array declared as **A**[*m*,*n*] as *m* groups of *n* items.)
35	**elem_size**	The packed element size, in bits, unused if the array is not packed.

Subrange Records. A subrange record describes a subrange type. From a subrange record, you can find

- Whether any auxiliary records modify the record.
- The name of the subrange type.
- Where in the source code the subrange type is defined.
- The data type of the members of the subrange.
- The lower bound of the subrange.
- The upper bound of the subrange.
- The size of the subrange.

Table 6-49 shows the format of a subrange record.

Table 6-49. Format of a Subrange Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_subrange** (decimal value 9), identifying the record as a subrange record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2-5	**noffset**	Byte offset from the beginning of the subrange record to the subrange type's name string.
6-9	**src_loc**	The source code location where the subrange type is defined.
10-13	**type_desc**	A type descriptor describing the data type of the subrange members.
14-17	**lo_bound**	The lowest value in the subrange.
18-21	**hi_bound**	The highest value in the subrange.
22-23	**size**	The size of the subrange, in bytes.

String Records. A string record describes a string type. From a string record you can find

- Whether any auxiliary records modify the record.
- The name of the string type.
- Where in the source code the string type is defined.
- The data type of the string elements.
- The data type of the string index.
- The lower bound of the string.
- The upper bound of the string.
- The size of the string.

Table 6-50 shows the format of a string record.

Table 6-50. Format of a String Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_string** (decimal value 38), identifying the record as a string record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**noffset**	Byte offset from the beginning of the string record to the string type's name string.
6–9	**src_loc**	The source code location where the string type is defined.
10–13	**elem_type_desc**	A type descriptor describing the data type of the string elements.
14–17	**indx_type_desc**	A type descriptor describing the data type of the string index.
18–21	**lo_bound**	The lowest index into the string.
22–25	**hi_bound**	The highest index into the string.
26–29	**size**	The size of a fixed length string, in bytes.

Set Records. A set record describes a set type. From a set record you can find

- Whether any auxiliary records modify the record.
- The name of the set type.
- Where in the source code the set type is defined.
- The data type of the set elements.
- The size of the set.

Table 6-51 shows the format of a set record.

Table 6-51. Format of a Set Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_set** (decimal value 10), identifying the record as a set record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**noffset**	Byte offset from the beginning of the set record to the set type's name string.
6–9	**src_loc**	The source code location where the set type is defined.
10–13	**type_desc**	A type descriptor describing the data type of the set elements.
14–15	**nbits**	The number of elements in the set.
16–17	**size**	The storage size of the set, in bytes.

Implicit Enumeration Records. An implicit enumeration record describes a Pascal-style enumerated type, that is, an enumerated type with implicit element values 0, 1, 2, and so on. From an implicit enumeration record, you can find

- Whether the record is followed by any auxiliary records.
- The name of the implicit enumeration type.
- Where in the source code the implicit enumeration type is defined.
- The number of elements in the enumeration.
- The size of the enumeration.
- The names of each of the enumerated elements.

Table 6-52 shows the format of an implicit enumeration record.

Table 6-52. Format of an Implicit Enumeration Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_implicit_enum** (decimal value 11), identifying the record as an implicit enumeration record.
1	**rec_flags**	**A set of flags. If bit** 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2-5	**noffset**	Byte offset from the beginning of the implicit enumeration record to the enumeration type's name string.
6-9	**src_loc**	The source code location where the enumeration type is defined.
10-11	**nelems**	The number of elements in the enumeration.
12-13	**size**	The size of the enumeration, in bytes.
14-end	**elem_noffset**	An array of byte offsets, each 4 bytes long, from the beginning of the implicit enumeration record to the name strings for each of the enumerated elements. The array elements appear in the same order as the enumerated elements. For example array element 0 is the offset to the name of enumerated element 0.

Explicit Enumeration Records. An explicit enumeration record describes a C-style enumerated type, that is, an enumerated type with explicit element values. From an explicit enumeration record you can find

- Whether the record is followed by any auxiliary records.
- The name of the explicit enumeration type.
- Where in the source code the explicit enumeration type is defined.
- The number of elements in the enumeration.
- The size of the enumeration.
- The names and values of each of the enumerated elements.

Table 6-53 shows the format of an explicit enumeration record.

Table 6-53. Format of an Explicit Enumeration Record

<table>
<tr><th>Byte Offset</th><th>Field Name</th><th>Value</th></tr>
<tr><td>0</td><td>rec_type</td><td>The enumerated constant dst_typ_explicit_enum (decimal value 12), identifying the record as an explicit enumeration record.</td></tr>
<tr><td>1</td><td>rec_flags</td><td>A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.</td></tr>
<tr><td>2–5</td><td>noffset</td><td>Byte offset from the beginning of the explicit enumeration record to the enumeration type's name string.</td></tr>
<tr><td>6–9</td><td>src_loc</td><td>The source code location where the enumeration type is defined.</td></tr>
<tr><td>10–11</td><td>nelems</td><td>The number of elements in the enumeration.</td></tr>
<tr><td>12–13</td><td>size</td><td>The size of the enumeration, in bytes.</td></tr>
<tr><td>14–end</td><td>files</td><td>An array of entries, one for each element in the enumeration. Each entry contains the following fields:
<table>
<tr><th>Byte Offset</th><th>Field Name</th><th>Value</th></tr>
<tr><td>0–3</td><td>noffset</td><td>Byte offset from the beginning of the explicit enumeration record to the name string for an enumerated element.</td></tr>
<tr><td>4–7</td><td>value</td><td>The value of the enumerated element.</td></tr>
</table>
</td></tr>
</table>

Record/Union Records. A record/union record describes a structure (also known as a record) or union type. From a record/union record, you can find

- Whether the data type is a structure or a union.
- Whether the record/union record is modified by any auxiliary records.
- The name of the structure or union type.
- The size of the structure or union type.
- The number of fields in the structure or union.
- The name of each field in the structure or union
- The data type of each field in the structure or union.
- The location of each field with respect to the beginning of the structure or union.

If a structure or union contains only **byte fields** (fields whose sizes are a whole number of bytes, located a whole number of bytes from the start of the structure or union), its record/union record uses the format shown in Table 6-54. Otherwise, its record/union record uses the format shown in Table 6-55. Table 6-56, Table 6-57, and Table 6-58 describe details of that format.

Table 6–54. Short Format of a Record/Union Record (for Records or Unions with Byte Fields Only)

<table>
<tr><th>Byte Offset</th><th>Field Name</th><th>Value</th></tr>
<tr><td>0</td><td>rec_type</td><td>The enumerated constant dst_typ_short_rec (decimal value 13) if the record describes a structure type, or the enumerated constant dst_typ_short_union (decimal value 15) if the record describes a union type.</td></tr>
<tr><td>1</td><td>rec_flags</td><td>A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.</td></tr>
<tr><td>2–5</td><td>noffset</td><td>Byte offset from the beginning of the record/union record to the structure or union type's name string.</td></tr>
<tr><td>6–9</td><td>src_loc</td><td>The source code location where the structure or union type is defined.</td></tr>
<tr><td>10–11</td><td>size</td><td>The size of the structure or union type, in bytes.</td></tr>
<tr><td>12–13</td><td>nfields</td><td>The number of fields in the structure or union type.</td></tr>
<tr><td>14–end</td><td>sfields</td><td>An array of elements, each of which describes one of the fields in the structure or union. Each array element contains the following fields:
<table>
<tr><th>Byte Offset</th><th>Field Name</th><th>Value</th></tr>
<tr><td>0–3</td><td>noffset</td><td>Byte offset from the beginning of the record/union record to the name string for the field.</td></tr>
<tr><td>4–7</td><td>type_desc</td><td>A type descriptor describing the data type of the field.</td></tr>
<tr><td>8–9</td><td>foffset</td><td>Byte offset from the beginning of the structure or union to the field.</td></tr>
</table></td></tr>
</table>

Table 6–55. General Format of a Record/Union Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_record** (decimal value 42) if the record describes a structure type, or the enumerated constant **dst_typ_union** (decimal value 43) if the record describes a union type.
1	**rec_flags**	**A set of flags. If bit** 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**noffset**	Byte offset from the beginning of the record/union record to the structure or union type's name string.
6–9	**src_loc**	The source code location where the structure or union type is defined.
10–11	**size**	The size of the structure or union type, in bytes.
12–13	**nfields**	The number of fields in the structure or union type.
14–end	**fields**	An array of elements, each of which describes one of the fields in the structure or union. Table 6–56 shows the format of an array element that describes a byte field. Table 6–57 shows the format of an array element that describes a bit field. Table 6–58 shows the format of an array element for any other type of field.

Table 6–56. Format of a **fields** *Array Element for a Byte Field*

Byte:Bit Offset	Field Name	Value
0–3	**noffset**	Byte offset from the beginning of the record/union type record to the name string for the field.
4–7	**type_desc**	A type descriptor describing the data type of the field.
11:1–11:0	**format_tag**	The enumerated value **dst_field_byte** (0) identifying the field as a byte field.
8:7–11:2	**offset**	Byte offset from the beginning of the structure or union to the beginning of the field.

Table 6–57. Format of a **fields** *Array Element for a Bit Field*

Byte:Bit Offset	Field Name	Value
0–3	**noffset**	Byte offset from the beginning of the record/union type record to the name string for the field.
4–7	**type_desc**	A type descriptor describing the data type of the field.
11:1–11:0	**format_tag**	The enumerated value **dst_field_bit** (1) identifying the field as a bit field.
11:7–11:2	**nbits**	The size of the field in bits.
10:0	**is_signed**	A Boolean value, true if the field value is signed.
10:3–10:1	**bit_offset**	Bit offset from the byte boundary preceding the field to the field.
8–9	**byte_offset**	Byte offset from the beginning of the structure or union to the byte boundary preceding the field.

Table 6–58. Format of a **fields** *Array Element for Any Other Type of Field*

Byte:Bit Offset	Field Name	Value
0–3	**noffset**	Byte offset from the beginning of the record/union type record to the name string for the field.
4–7	**type_desc**	A type descriptor describing the data type of the field.
11:1–11:0	**format_tag**	The enumerated value **dst_field_loc** (decimal value 2) identifying the field as a general field (neither a byte field nor a bit field).
8:7–11:2	**loffset**	Byte offset from the beginning of the record/union type record to a location string for the field.

File Records. A file record describes a variable of the Pascal **file** data type. From a file record, you can find

- Whether the record is followed by any auxiliary records.
- The name of the file type.
- Where in the source code the file type is defined.
- The data type of the values in the file.

Table 6-59 shows the format of a file record.

Table 6-59. Format of a File Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_file**, (decimal value 17), identifying the record as a file record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2-5	**noffset**	Byte offset from the beginning of the file record to the file type's name string.
6-9	**src_loc**	The source code location where the file type is defined.
10-13	**type_desc**	A type descriptor describing the data type of the values in the file.

Alias Records. An alias record describes a type alias, such as a C **typedef** or a Pascal defined TYPE. From an alias record, you can find

- Whether the record is followed by any auxiliary records.
- The name of the alias type.
- Where in the source code the alias type is defined.
- The data type that the alias renames.

Table 6-60 shows the format of an alias record.

Table 6-60. Format of an Alias Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_alias**, (decimal value 19), identifying the record as an alias record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2-5	**noffset**	Byte offset, from the beginning of the alias record, to the alias type's name string.
6-9	**src_loc**	The source code location where the alias type is defined.
10-13	**type_desc**	A type descriptor describing the data type that the alias renames.

Signature Records. A signature record describes a routine (procedure or function) type. From a signature record you can find

- Whether the record is followed by any auxiliary records.
- The name of the routine type.
- Where in the source code the routine type is defined.
- The symbol record for the routine's return value.
- The number of arguments that the routine takes.
- The symbol record for each of the routine's arguments.
- How each argument is passed.
- Whether each argument can be read by a routine called from this routine.
- Whether each argument can be written by a routine called from this routine.
- Whether each argument is visible to a routine called from this routine.

Table 6-61 shows the format of a signature record. Table 6-62 describes the flags used in a signature record.

Table 6-61. Format of a Signature Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_signature**, (decimal value 20), identifying the record as a signature record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**noffset**	Byte offset from the beginning of the signature record to the routine type's name string.
6–9	**src_loc**	The source code location where the routine type is defined.
10–13	**result**	Byte offset from the beginning of the signature record to the variable record for the routine's return value.
14–15	**nargs**	The number of arguments that the routine takes.
16–end	**args**	An array of elements, each of which describes one of the routine's arguments. Each element contains the following fields:

Byte Offset	Field Name	Value
0–3	**var_offset**	Byte offset from the beginning of the signature record to the variable record for the argument.
4–5	**attributes**	A set of flags describing how the argument is passed and whether it can be read by, written by, or visible to a called routine. Table 6-62 describes each flag. **attributes** is the logical OR of all the applicable flags.

Table 6–62. Flags in a Signature Record

Hex Value	Flag Name	Meaning
0x01	**DST_ARG_ATTR_VAL**	Passed by value.
0x02	**DST_ARG_ATTR_REF**	Passed by reference.
0x04	**DST_ARG_ATTR_NAME**	Passed by name.
0x08	**DST_ARG_ATTR_IN**	Can be read by a called routine.
0x10	**DST_ARG_ATTR_OUT**	Can be written by a called routine.
0x20	**DST_ARG_ATTR_HIDDEN**	Not visible in a called routine.

6.5.10.6 Auxiliary Records

A symbol record or type record may be modified by one or more auxiliary records. Auxiliary records, if used, immediately follow the symbol record or type record that they modify. An auxiliary record is one of the following:

Auxiliary Size Record. Modifies a variable record or a type record by overriding the default size of the variable or type. Table 6–63 shows the format of an auxiliary size record.

Table 6–63. Format of an Auxiliary Size Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_size**, (decimal value 30), identifying the record as an auxiliary size record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2–5	**size**	The size of the variable or type modified by this record.

Auxiliary Align Record. Modifies a variable record or a type record by overriding the default alignment of the variable or type. Table 6-64 shows the format of an auxiliary align record.

Table 6-64. Format of an Auxiliary Align Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_align**, (decimal value 31), identifying the record as an auxiliary align record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2-3	**alignment**	The number of low order zero bits in the address of the variable or type modified by this record. (For example, if an object is word-aligned, the last two bits of its address must be zero; the value of **alignment** is 2.)

Auxiliary Field Size Record. Modifies a record/union type record by overriding the default size for a field in the structure or union. Table 6-65 shows the format of an auxiliary field size record.

Table 6-65. Format of an Auxiliary Field Size Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_field_size**, (decimal value 32), identifying the record as an auxiliary field size record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2-3	**field_no**	The field number identifying the field whose size is modified by this record.
4-7	**size**	The size of the field modified by this record, in bits.

Auxiliary Field Offset Record. Modifies a record/union type record. It specifies the offset for a field when the offset is larger than 2^{16}. Table 6-66 shows the format of an auxiliary field offset record.

Table 6-66. Format of an Auxiliary Field Offset Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_field_off**, (decimal value 33), identifying the record as an auxiliary field offset record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2-3	**field_no**	The field number identifying the field whose offset if specified by this record.
4-7	**foffset**	Byte offset, from the beginning of the structure or union, to the field.

Auxiliary Field Align Record. Modifies a record/union type record by overriding the default alignment for a field. Table 6-67 shows the format of an auxiliary field align record.

Table 6-67. Format of an Auxiliary Field Align Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_field_align**, (decimal value 34), identifying the record as an auxiliary field align record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2-3	**field_no**	The field number identifying the field whose alignment is specified by this record.
4-5	**alignment**	The number of low order zero bits in the field's address. (For example, if the field is word-aligned, the last two bits of its address must be zero; the value of **alignment** is 2.)

Auxiliary Var Bound Record. Modifies an array record, a subrange record, or a string record. It defines the lower or upper bound of an array, subrange, or string whose bounds are not known at compile time. Table 6-68 shows the format of an auxiliary var bound record.

Table 6-68. Format of an Auxiliary Var Bound Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_var_bound**, (decimal value 36), identifying the record as an auxiliary var bound record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2-3	**which**	The enumerated constant **dst_low_bound** (0) if this record specifies a lower bound, or the enumerated constant **dst_high_bound** (1) if this record specifies an upper bound.
4-7	**voffset**	Byte offset from the beginning of the auxiliary var bound record to the symbol record for the variable that stores the bound.

Auxiliary Variable Lifetime Record. Modifies a variable record. On a Series 10000 workstation, compilers may choose to create separate, overlapping lifetimes for a single variable; the compiler must store the variable in two different locations corresponding to its two overlapping lifetimes. An auxiliary variable lifetime record adds the location information for an additional variable lifetime. Table 6-69 shows the format of an auxiliary variable lifetime record.

Table 6-69. Format of an Auxiliary Variable Lifetime Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_lifetime**, (decimal value 49), identifying the record as an auxiliary variable lifetime record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2-5	**loffset**	Byte offset from the beginning of the auxiliary variable lifetime record to the variable record for the additional lifetime.

Auxiliary Type Derivation Record. Modifies a type record for a C++ inherited type. In C++, you may define a type in terms of a parent type: the child type includes all the fields in the parent type and any additional fields in the child type declaration. The auxiliary type derivation record identifies the parent type. The record describing the child type must include all the fields in the parent type; the auxiliary type derivation record does not provide this information. Table 6–70 shows the format of an auxiliary type derivation record.

Table 6–70. Format of an Auxiliary Type Derivation Record

Byte:Bit Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_type_deriv**, (decimal value 44), identifying the record as an auxiliary type derivation record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2–5	**parent_type**	A type descriptor describing the parent type.
6–7	**orig_field_no**	A field number within the record/union record. The field specifies the location of the inherited record.
8:0	**orig_is_pointer**	A Boolean value. If **orig_is_pointer** is true, the field identified by **orig_field_no** is a pointer to the inherited record. If **orig_is_pointer** is false, the inherited record begins at the field identified by **orig_field_no**.

Auxiliary Pointer Base Record. Modifies a variable record for a FORTRAN pointer-based variable. It points to the variable record whose value is the location of the pointer-based variable. Table 6–71 shows the format of an auxiliary pointer base record.

Table 6–71. Format of an Auxiliary Pointer Base Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_ptr_base**, (decimal value 50), identifying the record as an auxiliary pointer base record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2–5	**voffset**	Byte offset from the beginning of the auxiliary pointer base record to the variable record whose value is the location of the pointer-based variable.

Auxiliary Register Value Record. Modifies an entry record by specifying a register and a value. Before jumping to the entry point described by the entry record, the debugger must set the specified register to the specified value. Table 6–72 shows the format of an auxiliary register value record.

Table 6–72. Format of an Auxiliary Register Value Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_aux_reg_val**, (decimal value 52), identifying the record as an auxiliary register value record.
1	**rec_flags**	A set of flags. If bit 1 (the second lowest bit) is set, the record is the last of a series of one or more auxiliary records.
2–3	**reg**	An enumerated value representing the register that the debugger must set before jumping to the entry point. The available values are listed in the file **/usr/include/apollo/isp.h**.
4–7	**loffset**	Byte offset from the beginning of the auxiliary pointer base record to a location string. The debugger must set the register specified by **reg** to the address to which the location string resolves. If the location string resolves to a register–relative address, the debugger can't jump to the entry point unless the current stack frame is the lexical parent of the block containing the entry point.

6.5.10.7 String Tables

The string tables contain the strings to which other records in the **.symbols** section refer. Table 6–31 shows the format of a string table.

6.5.10.8 Pad Records

Like the **.blocks** section, **.symbols** may include pad records. A pad record contains no data, but provides two bytes of padding, for alignment purposes. The format of a pad record in **.symbols** is the same as the format of a pad record in **.blocks**, as shown in Table 6–37.

6.5.10.9 Forward Records

A forward record is a reference to another record. Instead of repeating identical information, a record can simply point to another record that contains the information. Table 6–73 shows the format of a forward record.

Table 6–73. Format of a Forward Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_forward**, (decimal value 29), identifying the record as a forward record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**rec_off**	Byte offset from the beginning of the forward record to the record containing the information.

6.5.10.10 Extension Records

Like the **.blocks** section, **.symbols** may include extension records. We provide extension records to allow compilers to generate records other than the ones defined in this section. Domain/DDE and **dbx** ignore these records. The format of an extension record in **.symbols** is the same as the format of an extension record in **.blocks**, as shown in Table 6–38.

6.5.10.11 End Scope Records

An end scope record marks the end of the symbol information for a single scope. Table 6–74 shows the format of an end scope record.

Table 6–74. Format of an End Scope Record

Byte Offset	Field Name	Value
0	**rec_type**	The enumerated constant **dst_typ_end_scope**, (decimal value 24), identifying the record as an end scope record.
1	**rec_flags**	A set of flags. If bit 0 (the low bit) is set, the record is modified by one or more auxiliary records.
2–5	**rec_off**	Byte offset from the beginning of the forward record to the record containing the information.

6.5.10.12 Source Code Locations

Many records in the **.symbols** section include the source code locations where symbols are defined. A source code location is a file and line number in the source code. Table 6-75 shows the format of a source code location.

Table 6-75. Format of a Source Code Location

Byte:Bit Offset	Field Name	Value
2:3-3:1	**file_index**	An index into the **.blocks** file table, identifying the source code file.
0:7-2:4	**line_number**	A line number within the source file.

6.5.10.13 Location Strings

The **.symbols** section represents machine locations, such as function entry points, or the location of a variable's value, by **location strings**. A location string is a sequence of bytes, ending in a zero byte. Some of the bytes in a location string contain location opcodes. The rest of the bytes contain operands to the opcodes.

A location opcode takes at most one operand; some opcodes take no operands. The operand, if one exists, immediately follows the opcode.

The length of its operand is part of an opcode's definition. Opcodes can take operands of one to four bytes, or can store their operands in the last three bits of the opcode byte.

To read a location string, the debugger acts as an interpreter. It creates an accumulator stack on which to interpret the string. On the accumulator stack, the debugger performs the operations specified by the opcodes. When the debugger has processed all the opcodes, the top of the accumulator stack contains either the location specified by the location string, or the name of the register that contains that address.

The sample location string in Figure 6-8 illustrates a 5-byte location string that specifies the location with address 06.

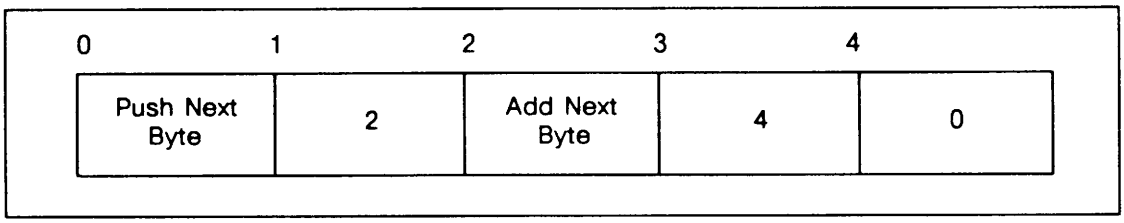


Figure 6-8. A Sample Location String.

Location Opcodes. In Figure 6-8, "Push Next Byte" and "Add Next Byte" are both **location opcodes**. The zero byte marks the end of the string.

Table 6-76 describes all of the location opcodes and their operands.

Table 6-76. Location Opcodes

Register Opcodes: The location is in register *n*, where *n* is the value in the operand. (Register numbers are enumerated values defined in **/usr/include/apollo/isp.h**.)

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
REGADR	10	✓				
REGADR1	6D		✓			
REGADR2	6E			✓		

Push-Register Opcodes: Push the contents of register *n* onto the accumulator stack, where *n* is the value in the operand. (Register numbers are enumerated values defined in **/usr/include/apollo/isp.h**.)

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
REG	18	✓				
REG1	70		✓			
REG2	71			✓		

Push-Section-Base Opcodes: Push the virtual address stored at index *n* (origin 0) of the current compilation unit's **.blocks** section table, where *n* is the value in the operand.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
SECT	20	✓				
SECT1	58		✓			
SECT2	59			✓		
SECT3	5a				✓	
SECT4	5b					✓

(Continued)

Table 6-76. Location Opcodes (Cont.)

Push-Constant Opcodes: Push the value in the operand.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
CONST	28	✓				
CONST1	5C		✓			
CONST2	5D			✓		
CONST3	5E				✓	
CONST4	5F					✓

Push-Negative Opcodes: Push the arithmetic negative of the value in the operand.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
NCONST	30	✓				
NCONST1	61		✓			
NCONST2	62			✓		
NCONST3	63				✓	
NCONST4	64					✓

Add-Constant Opcodes: Add the value in the operand to the top of the stack. Replace the top of the stack with the sum.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
ADDC	38	✓				
ADDC1	65		✓			
ADDC2	66			✓		
ADDC3	67				✓	
ADDC4	68					✓

(Continued)

Table 6-76. Location Opcodes (Cont.)

Subtract-Constant Opcodes: Subtract the value in the operand from the top of the stack. Replace the top of the stack with the result.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
SUBC	40	✓				
SUBC1	69		✓			
SUBC2	6a			✓		
SUBC3	6b				✓	
SUBC4	6c					✓

Opcode Name	Hex Value	Opcode Operation
ADD	09	Pop two values from the top of the stack, add them, and push the result.
NEGATE	0A	Arithmetically negate the top of the stack.
LSHIFT	0B	Shift the top of the stack left by *n* bits, where *n* is the value in the next byte.
EXTB	0C	Sign extend the low byte of the top of the stack.
EXTW	0D	Sign extend the low word of the top of the stack.
INDIRECT	0E	Treat the top of the stack as an address, *a1*. Treat the value at a1 as another address, *a2*. Replace the top of the stack with the address *a2*.
PC	72	Push the PC.
NOP	02	Do nothing.
UNKNOWN	01	Cause "unrepresentable location" error.
NULL	00	End a location string.
VARIABLE POINTER BASE	B8	Push the value of the pointer variable referenced by the *n*th auxiliary pointer base record following this record, where *n* is the value in the low 3 bits of the opcode byte.

Range Opcodes. A code optimizer may store a variable in several locations. The variable's current location is determined by which portion of the code is executing. A location string, therefore, may consist of several **substrings**, each of which specifies a location of the variable.

A **range opcode** marks the beginning of a substring. The range opcode and its operand identify a PC range or a range of source code lines. Within that range, the location specified in the substring is valid.

The **flags** field in the **.blocks** header determines whether the block's range operands are expressed in PCs or in source code lines. On an MC680x0-based workstation, a range operand is either a number of source code lines or a number of PC locations multiplied by two. On a Series 10000 workstation, a range operand is either a number of source code lines or a number of PC locations multiplied by four.

To save space, a range opcode doesn't always specify the beginning and the end of the range. A range opcode specifies the length of the range. Ordinarily, the beginning of the range is one of the following:

- The end of the previous range, if this is not the first range specified in the string
- The beginning of the current block, if this is the first range specified in the string and ranges are in PCs
- The beginning of the file, if this is the first range specified in the string and ranges are in source code lines

Several **range modifier opcodes** may appear before a range opcode.

- A **range delta opcode** moves the beginning of the range by a specified number of PC locations or source code lines. A range delta opcode can move the beginning of the range either forward or backward.
- A **file state opcode** changes the current file. If the file state opcode appears before a range delta opcode, the file state opcode moves the beginning of the range to the beginning of the specified file. Otherwise, it moves the end of the range to the specified file. File state opcodes appear only for ranges measured in source code lines.
- A **statement escape opcode** changes the statement number from its default value, zero. If the statement escape opcode appears before a range delta opcode, the statement escape opcode sets the statement number for the beginning of the range. Otherwise, it sets the statement number for the end of the range. Statement escape opcodes appear only for ranges measured in source code lines.

In the example substring illustrated in Figure 6-9, range operands specify the number of source code lines. The location 1000 is valid from file number 5, source line 10, statement 1 to file number 5, source line 35 (25 lines from the beginning of the range), statement 2.

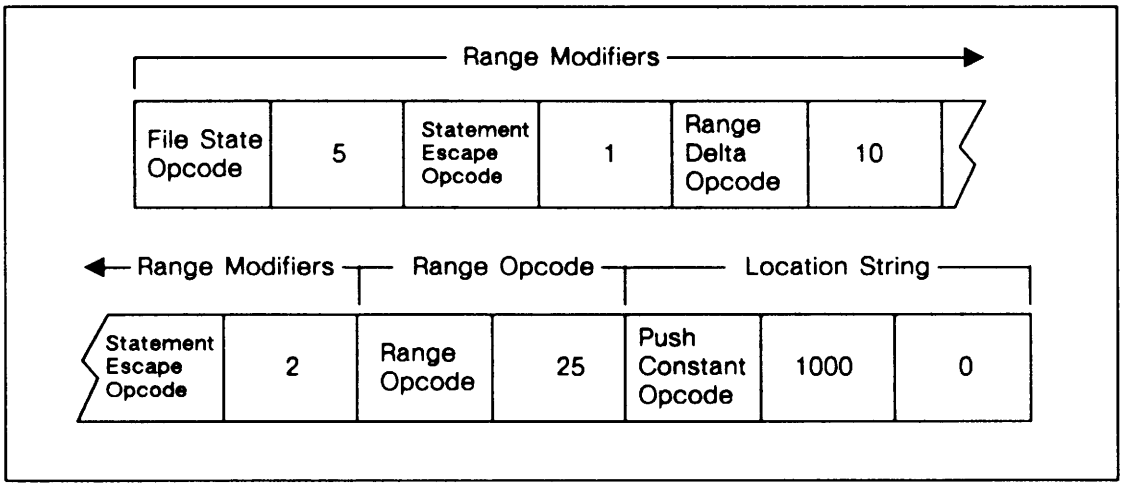


Figure 6-9. A Sample Location Substring

The first substring in a location string need not have a range opcode. By default, the range for the first substring is all portions of the code that have not been covered by another substring within the location string.

The sample location string in Figure 6-10 belongs to a Series 10000 workstation. Range operands specify the number of PC locations multiplied by 4. From the beginning of the current block to offset 32, the location is 200. Otherwise, the location is 3.

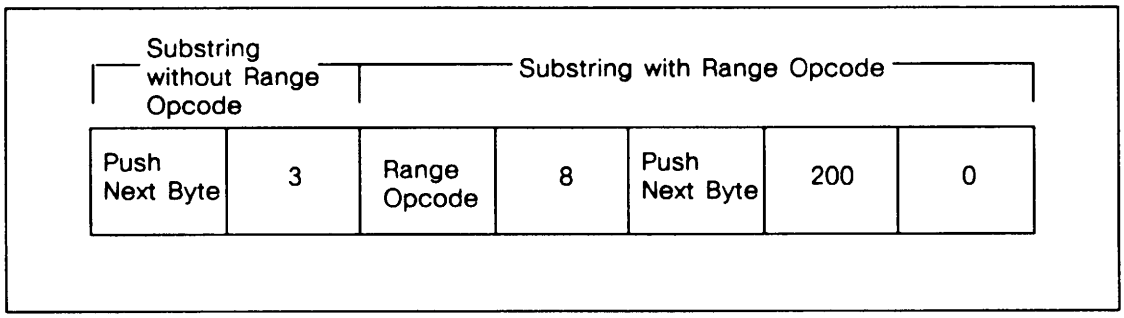


Figure 6-10. A Location String Composed of Two Substrings

Table 6-77 describes the range opcodes and range modifier opcodes.

Table 6–77. Range Opcodes and Range Modifier Opcodes

Range Opcodes: The range is *n* PC locations or source lines long, where *n* is the value in the operand.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
RSIZE	A0	✓				
RSIZE1	88		✓			
RSIZE2	89			✓		
RSIZE3	8A				✓	
RSIZE4	8B					✓

Range Delta Opcodes: Move the beginning of the range forward by *n* PC locations or source lines, where *n* is the value in the operand.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
RDELTA	90	✓				
RDELTA1	80		✓			
RDELTA2	81			✓		
RDELTA3	82				✓	
RDELTA4	83					✓

Negative Range Delta Opcodes: Move the beginning of the range backward by *n* PC locations or source lines, where *n* is the value in the operand.

Opcode Name	Hex Value	Operand				
		Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
RNDELT	98	✓				
RNDELT1	84		✓			
RNDELT2	85			✓		
RNDELT3	86				✓	
RNDELT4	87					✓

(Continued)

Table 6-77. Range Opcodes and Range Modifier Opcodes (Cont.)

File State Opcodes: Change to file number *n*, where *n* is the value in the operand.						
		Operand				
Opcode Name	**Hex Value**	Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
FILE	A8	✓				
FILE1	8D		✓			
FILE2	8E			✓		

Statement Escape Opcodes: Change to statement number *n*, where *n* is the value in the operand.						
		Operand				
Opcode Name	**Hex Value**	Low 3 Bits	Next Byte	Next 2 Bytes	Next 3 Bytes	Next 4 Bytes
STMT	B0	✓				
STMT1	8F		✓			

Since a zero byte marks the end of a location string, we provide **zero opcodes** for representing values that contains zero bytes. A zero opcode left-shifts the preceding value, then replaces the low byte(s) of the value with the operand. To represent the value 0x1200, for example, you would use a zero opcode to shift the value 0x12 leftward one byte. To represent the value 0x12003456, you would use a zero opcode to shift the value 0x12 leftward three bytes, then replace the low two bytes of the value with 0x34 0x56. Table 6-78 describes the zero opcodes.

Table 6-78. Zero Opcodes

Opcode Name	**Hex Value**	**Opcode Operation**
ZERO1	73	Left-shift the preceding value one byte.
ZERO2 ADD1	74	Left-shift the preceding value two bytes. Replace the value's low byte with the next byte.
ZERO3 ADD1	75	Left-shift the preceding value one byte. Replace the value's low byte with the next byte.
ZERO3 ADD2	76	Left-shift the preceding value two bytes. Replace the value's low two bytes with the next two bytes.

6.5.11 The .lines Section

The **.lines** section is the third of the debugging sections. Each block defined in the **.blocks** section has its own table in the **.lines** section. An entry in a **.lines** table maps a source code line to a range of machine instructions.

For compactness, **.lines** stores **line number deltas** and **PC deltas**, rather than absolute line numbers and instruction addresses. The first entry in the table is an absolute line number and an absolute address of an instruction. Subsequent entries store the difference between the current line number and the previous entry's line number, and the difference between the current instruction address and the previous entry's instruction address. Entries in **.lines** are only one byte long: the high four bits are the line number delta, which can store values from –7 to 7; the low four bits are the PC delta, which can store values from 0 to 15.

Escape sequences handle entries that can't be accommodated by this simple delta scheme. A value of 0x8 in the high four bits of the byte signifies an escape entry. The low four bits of the byte contain an escape function code. Escape entries are usually followed by additional data, specific to the escape function. Table 6–79 describes the escape function codes.

Table 6-79. Escape Function Codes

Decimal Value	Escape Code Name	Meaning
0	**DST_LN_PAD**	This byte is padding.
1	**DST_LN_FILE**	The file and line number for the next table entry is the source code location specified in the next four bytes.
2	**DST_DLN1_DPC1**	The next table entry is a 1-byte line delta followed by a 1-byte PC delta.
3	**DST_LN_DLN2_DPC2**	The next table entry is a 2-byte line delta followed by a 2-byte PC delta.
4	**DST_LN_LN4_PC4**	The next table entry is a 4-byte absolute line number followed by a four-byte absolute PC.
5	**DST_LN_DLN1_DPC0**	The next table entry is a 1-byte line delta. The PC delta is zero.
6	**DST_LN_LN_OFF_1**	The next table entry refers to the second statement on the specified source code line.
7	**DST_LN_LN_OFF**	The next table entry refers to the *n*th statement on the specified source code line, where *n* is 1 plus the value in the next byte.
8	**DST_LN_ENTRY**	This code instructs the debugger to insert a breakpoint at the source code line specified in the next table entry.*
9	**DST_LN_EXIT**	The next table entry is the last position within the current block before the end of a procedure.
14	**DST_LN_NXT_BYTE**	The next byte contains the escape code.
15	**DST_LN_END**	This code marks the end of the line number table. It is followed by a single entry whose low four bits contain the code size of the last statement in the block.

* Breakpoints should appear only after the function "prologue," that is, after the code that pushes the stack frame and allocates local variables.

6.6 Relocation Information

Both the linker and the loader use relocation information to identify the references that they must change. The linkers and the loader may also use relocation entries to resolve external references.

Relocation information contains one entry for each relocatable reference. From a relocation entry, you can find

- The virtual address of the relocatable reference.
- The section containing the referenced item. From this information, the linker or loader can determine the virtual address of the beginning of the section, both before and after linking or loading.

Relocation entries are grouped by the section in which the reference appears (not the section in which the referenced item appears).

The **coffdump** or **dump** command with the **-r** option displays the relocation information.

Table 6-80 shows the format of a relocation entry.

Table 6-80. Format of a Relocation Entry

Byte Offset	Field Name	Value
0-3	**r_vaddr**	The virtual address of the relocatable reference or, if **r_type** is **R_NRELOC**, the number of relocatable references in the section.
4-7	**r_symndx**	The symbol table index for a symbol in the section, usually the referenced item or section name. The symbol table entry identifies the symbol being referenced.
8-9	**r_type**	One of the following enumerated values: 0 **R_ABS** No relocation is necessary. 6 **R_DIR32** The reference is a 4-byte virtual address. 512 **R_NRELOC** **r_vaddr** does not contain the reference address; instead, it contains the number of relocatable references in the section. The compiler uses **r_vaddr** this way if the number of references is too large for the **s_nreloc** field in the section header. **R_NRELOC** is available only in the first relocation entry for a section, in Apollo COFF files only.

6.7 Line Number Tables

Any COFF file may contain a line number table. A line number table maps source code line numbers to the addresses of the corresponding machine instructions, and is grouped by procedures. The first entry in a procedure grouping has line number zero, and the symbol table index for the procedure name in place of a virtual address. Subsequent entries allow you to find

- The source code line number for the code that this entry describes.
- The virtual starting address of the machine instruction(s) corresponding to the source code.

The **coffdump** or **dump** command with the **-l** option displays the line number tables.

Table 6-81 shows the format of of a line number entry.

Table 6-81. Format of a Line Number Entry

Byte Offset	Field Name	Value
0-3	**l_paddr** or **l_symindx**	The virtual starting address of the machine instruction(s) corresponding to the source code described by this entry or the symbol table index of the procedure name.
4-5	**l_lnno**	The source code line number for the code described by this entry. The line number is zero for the first entry in a procedure grouping.

NOTE: In Apollo COFF files, the information in the **.lines** section (described in Section 6.5.11) supersedes the information in the line number table. For compatibility with the AT&T COFF template, however, all Domain compilers except the SR10.0.p *PRISM* compilers will create line number tables, as well as the **.lines** section, when invoked with one of the full debugging options.

6.8 Symbol Table

Any COFF file may have a symbol table. The symbol table stores information about a symbol, including its name, data type, and value.

In the AT&T COFF template, a symbol table may store entries for each file, each function within a file, each symbol in each module, and each COFF section name, as well as some special symbols described in Section 6.8.6.

In the Apollo implementation of COFF, much of the symbol information is found in the **.symbols** section, described in Section 6.5.10. The symbol tables in Apollo COFF files contain only the symbol information required by the linkers and the loader These are entries for filenames, functions, section names, global symbols, and the special symbols **stext**, **etext**, **edata**, and **end**. Symbols appear in the order shown in Figure 6-11.

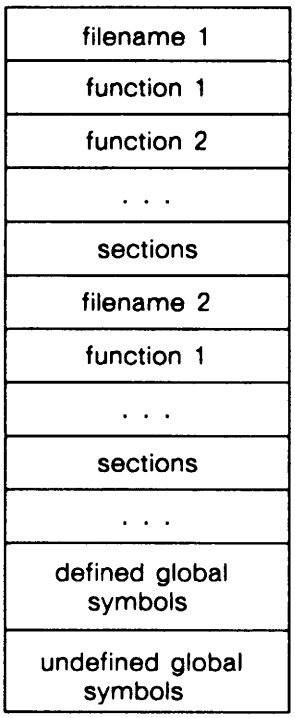


Figure 6-11. Apollo COFF Symbol Table

In the following sections, we first describe the general format for symbol table entries, defined by the COFF template. Next, we describe the symbol table entries found in Apollo COFF files. The Apollo symbol table entries are all specific instances of the general format.

The **coffdump** or **dump** command with the **-t** option displays the symbol table.

6.8.1 Format for Symbol Table Entries

The symbol table represents a symbol by a **main entry** and, for some symbols, one or more **auxiliary entries**. The COFF template specifies a general format for a symbol's main entry, summarized in Table 6-82. Table 6-83, Table 6-84, Table 6-85, and Table 6-86 detail specific fields in the main entry format. Symbol table entries each contain 18 bytes. Since symbol table entry indices begin at zero, and auxiliary entries count as one symbol each, byte offsets may be readily calculated. Auxiliary entries do not follow any general format, although they must also have 18 bytes each.

In Sections 6.8.2 through 6.8.6, we describe the formats of the main and auxiliary entries of filenames, functions, sections, global symbols, and special symbols.

Table 6-82. General Format for Any Symbol's Main Entry

Byte Offset	Field Name	Value
0-7	**_n**	The symbol's name, if it has fewer than eight characters. Names of more than eight characters are stored in the string table. In that case, the first four bytes of **_n** are zero, and the last four bytes contain the offset of the symbol name from the beginning of the string table. Section 6.9 describes the string table.
8-11	**n_value**	The symbol's value. The way in which the symbol's value is specified depends on the symbol's storage class.
12-13	**n_scnum**	The section number of the section containing the symbol, or one of the enumerated constants listed in Table 6-83.
14-15	**n_type**	The least significant 4 bits of this field (bits 0-3) contain an enumerated constant identifying the symbol's basic (fundamental) data type. Table 6-84 lists the available values. The remaining 12 bits are divided into 6 2-bit fields. These fields represent the symbol's first through sixth derived types, in order: bits 4-5 represent the first derived type; bits 11-12 represent the sixth derived type. Each field contains one of the enumerated constants listed in Table 6-85.
16	**n_sclass**	An enumerated constant identifying the symbol's storage class. Table 6-86 lists the available values.
17	**n_numaux**	The number of auxiliary entries following this entry.*
* Although the AT&T COFF template allows multiple auxiliary entries for any main entry, many programs assume that no symbol will have more than one auxiliary entry.		

Table 6–83. Enumerated Constants Available for the **n_scnum** *Field*

Decimal Value	Constant Name	Meaning
–2	**N_DEBUG**	The symbol is a special debugging symbol.
–1	**N_ABS**	The symbol is an absolute symbol, that is, its virtual address cannot be changed.
0	**N_UNDEF**	The symbol is an undefined external.

Table 6–84. Enumerated Constants Identifying a Symbol's Basic Data Type

Decimal Value	Constant Name	Data Type
0	**T_NULL**	Type not assigned
1	**T_VOID**	Void
2	**T_CHAR**	Character
3	**T_SHORT**	Short integer
4	**T_INT**	Integer
5	**T_LONG**	Long integer
6	**T_FLOAT**	Floating–point
7	**T_DOUBLE**	Double word
8	**T_STRUCT**	Structure
9	**T_UNION**	Union
10	**T_ENUM**	Enumeration
11	**T_MOE**	Member of enumeration
12	**T_UCHAR**	Unsigned character
13	**T_USHORT**	Unsigned short
14	**T_UINT**	Unsigned integer
15	**T_ULONG**	Unsigned long

Table 6–85. Enumerated Constants Identifying a Symbol's Derived Type(s)

Decimal Value	Constant Name	Derived Type
0	**DT_NON**	No derived type
1	**DT_PTR**	Pointer
2	**DT_FCN**	Function
3	**DT_ARY**	Array

Table 6–86. Enumerated Constants Identifying a Symbol's Storage Class

Decimal Value	Constant Name	Storage Class
–1	**C_EFCN**	Physical end of function
0	**C_NULL**	No storage class
1	**C_AUTO**	Automatic variable
2	**C_EXT**	External symbol
3	**C_STAT**	Static
4	**C_REG**	Register variable
5	**C_EXTDEF**	External definition
6	**C_LABEL**	Label
7	**C_ULABEL**	Undefined label
8	**C_MOS**	Member of structure
9	**C_ARG**	Function argument
10	**C_STRTAG**	Structure tag
11	**C_MOU**	Member of union
12	**C_UNTAG**	Union tag
13	**C_TPDEF**	Type definition
14	**C_USTATIC**	Uninitialized static
15	**C_ENTAG**	Enumeration tag

(Continued)

Table 6-86. Enumerated Constants Identifying a Symbol's Storage Class (Cont.)

Decimal Value	Constant Name	Storage Class
16	C_MOE	Member of enumeration
17	C_REGPARM	Register parameter
18	C_FIELD	Bit field
100	C_BLOCK	Beginning or end of block
101	C_FCN	Beginning or end of function
102	C_EOS	End of structure
103	C_FILE	Filename
104	C_LINE	Used only by utility programs
105	C_ALIAS	Duplicated tag
106	C_HIDDEN	Like static, used to avoid name conflicts
110	C_EXT_UNMARKED	Deleted. This value is available in Apollo COFF files only.
111	C_EXT_EXPUNGED	Ignored. This value is available in Apollo COFF files only.

6.8.2 Filename Entries

The symbol table represents a filename by a main entry and an auxiliary entry. From a filename's symbol table entries, you can find:

- The symbol table index for the next filename entry.
- The complete pathname for the file.

NOTE: The AT&T COFF template specifies that the symbol table store only the filename. In Apollo COFF files, the symbol table stores the complete pathname.

Table 6-87 and Table 6-88 show the format of a filename's symbol table entries.

Table 6-87. Format of a Filename's Main Symbol Table Entry

Byte Offset	Field Name	Value
0-7	**_n**	The character string ".file" (padded to 8 bytes with trailing zeros), identifying the entry as a filename main entry.
8-11	**n_value**	The symbol table index for the next filename main entry, or if this is the last filename main entry, the symbol table index of the first global symbol.
12-13	**n_scnum**	-2
14-15	**n_type**	The enumerated constant **T_NULL** (decimal value 0).
16	**n_sclass**	The enumerated value **C_FILE** (decimal value 103).
17	**n_numaux**	1, the number of auxiliary records for this entry.

Table 6-88. Format of a Filename's Auxiliary Symbol Table Entry

Byte Offset	Field Name	Value
0-13	**x_file.x_fname**	The full pathname of the file, padded with trailing zeros, if necessary. If the full pathname of the file is longer than 14 characters, **x_file.x_fname** contains 4 bytes of zeros followed by the byte offset from the beginning of the COFF string table to the pathname of the file. This is an Apollo extension to AT&T COFF.

6.8.3 Function Entries

The symbol table represents a locally-defined function by a main entry and an auxiliary entry. (The symbol table represents calls to externally-defined functions in the same way that it represents other global symbols, as described in Section 6.8.5.) From a function's symbol table entries, you can find

- The name of the function.
- The virtual address of the top of the function.
- The COFF file section containing the function's machine code.
- Whether these entries represent a reference to an externally defined function.

- The size of the function.
- The line number entries for the function.
- The symbol table index of the next entry following the end of the function.

Table 6-89 and Table 6-90 show the formats of a function's symbol table entries.

Table 6-89. Format of a Function Name's Main Symbol Table Entry

Byte:Bit* Offset	Field Name	Value
0-7	**_n**	The function's name, padded with trailing zeros if necessary. If the function's name is longer than 8 characters, **_n** contains 4 bytes of zeros followed by the byte offset from the beginning of the COFF string table to the function name's entry.
8-11	**n_value**	The virtual address of the top of the function. Note that the top of the function is not necessarily the function's entry point.
12-13	**n_scnum**	The section number of the section containing the function. (Sections are numbered, from 1, in the order in which they appear in the table of section headers.)
15:0-15:3	**typ**	The enumerated constant **T_INT** (decimal value 4).
15:4-15:5	**d1**	The enumerated constant **DT_FCN** (decimal value 2) identifying the symbol as a function.
16	**n_sclass**	The enumerated constant **C_EXT** (decimal value 2) or **C_STAT** (decimal value 3).
17	**n_numaux**	1, the number of auxiliary records for this entry.

Table 6-90. Format of a Function Name's Auxiliary Symbol Table Entry

Byte Offset	Field Name	Value
0-3	**x_misc.x_tagndx**	0
4-7	**x_misc.x_fsize**	The size of the function in bytes.
8-11	**x_sym.x_fcn.x_lnnoptr**	Byte offset from the beginning of the COFF file to the line number entries for this function.
12-15	**x_sym.x_fcn.x_endndx**	Symbol table index of the next entry following the end of the function.

6.8.4 Section Entries

The symbol table represents a section name by a main entry and an auxiliary entry. From a section's symbol table entries, you can find

- The name of the section.
- The virtual address of the beginning of the section.
- The section number. (Sections are numbered, from 1, in the order in which they appear in the table of section headers.)
- The size of the section.
- The number of relocation entries for this section.
- The number of line number entries for this section.

Table 6-91 and Table 6-92 show the format of a section's symbol table entry.

Table 6-91. Format of a Section's Main Symbol Table Entry

Byte Offset	Field Name	Value
0-7	**_n**	The section's name, padded with trailing zeros if necessary. If the section's name is longer than 8 characters, **_n** contains 4 bytes of zeros followed by the byte offset, from the beginning of the COFF string table, to the section's name.
8-11	**n_value**	The virtual address of the beginning of the section.
12-13	**n_scnum**	The section number. (Sections are numbered, from 1, in the order in which they appear in the table of section headers.)
14-15	**n_type**	The enumerated constant **T_NULL** (decimal value 0).
16	**n_sclass**	The enumerated constant **C_STAT** (decimal value 3).
17	**n_numaux**	1, the number of auxiliary records for this entry.

Table 6–92. Format of a Section's Auxiliary Symbol Table Entry

Byte Offset	Field Name	Value
0–3	**x_scn.x_scnlen**	The number of bytes that the loader allocates for the section.
4–5	**x_scn.x_nreloc**	The number of relocation entries for the section.
6–7	**x_scn.x_nlinno**	The number of line number entries for the section.

6.8.5 Global Symbol Entries

A global symbol is represented in the symbol table by a main entry without an auxiliary entry. From a global's symbol table entry, you can find

- The name of the global.
- The virtual address of the global, if it is initialized.
- The size of the global, if it is uninitialized.
- Whether the global is defined externally.
- Whether the global is uninitialized.
- The COFF file section containing the global's definition, if it is initialized.
- The global's data type.

Table 6–93 shows the format of a global's symbol table entry.

Table 6-93. Format of a Global's Symbol Table Entry

Byte Offset	Field Name	Value
0-7	**_n**	The global symbol's name, padded with trailing zeros if necessary. If the global's name is longer than eight characters, **_n** contains 4 bytes of zeros followed by the byte offset from the beginning of the COFF string table to the global's name.
8-11	**n_value**	The virtual address of the global if it is initialized, or the size of the global if it is uninitialized, or 0 if the global is defined externally.
12-13	**n_scnum**	The section number of the section in which the global is defined, or 0 if the global is uninitialized or external. (Sections are numbered, from 1, in the order in which they appear in the table of section headers.)
14-15	**n_type**	The least significant 4 bits of this field (bits 0-3) contain an enumerated constant identifying the symbol's basic (fundamental) data type. Table 6-84 lists the available values. The remaining 12 bits are divided into 6 2-bit fields. These fields represent the symbol's first through sixth derived types, in order: bits 4-5 represent the first derived type; bits 11-12 represent the sixth derived type. Each field contains one of the enumerated constants listed in Table 6-85.
16	**n_sclass**	The enumerated value **C_EXT** (decimal value 2) or **C_EXT_UNMARKED** (decimal value 110) or **C_EXT_EXPUNGED** (decimal value 111). (Linker options unmark or expunge symbols.)
17	**n_numaux**	0, the number of auxiliary records for this entry.

6.8.6 Special Symbols

According to the AT&T COFF template, a symbol table may store entries for several special symbols that help organize the symbol table. Apollo COFF files created by **ld** store symbol table entries for special symbols that mark the beginning and end of the **.text** section, the end of the **.data** section, and the end of the **.bss** section. (Domain compilers, assemblers, and **bind** do not create entries for these symbols.)

6.8.6.1 The stext Symbol

The **stext** symbol marks the beginning of the **.text** section. Table 6-94 shows the format of the symbol table entry for **stext**.

Table 6-94. Format of the Symbol Table Entry for **stext**

Byte Offset	Field Name	Value
0-7	**_n**	The character string "stext" (padded to eight bytes with trailing zeros).
8-11	**n_value**	The virtual address of the beginning of the **.text** section.
12-13	**n_scnum**	-1
14-15	**n_type**	The enumerated constant **T_NULL** (decimal value 0).
16	**n_sclass**	The enumerated constant **C_EXT** (decimal value 2).
17	**n_numaux**	0, the number of auxiliary records for this entry.

6.8.6.2 The etext Symbol

The **etext** symbol marks the end of the **.text** section. Table 6-95 shows the format of the symbol table entry for **etext**.

Table 6-95. Format of the Symbol Table Entry for **etext**

Byte Offset	Field Name	Value
0-7	**_n**	The character string "etext" (padded to eight bytes with trailing zeros).
8-11	**n_value**	The virtual address of the end of the **.text** section.
12-13	**n_scnum**	-1
14-15	**n_type**	The enumerated constant **T_NULL** (decimal value 0).
16	**n_sclass**	The enumerated constant **C_EXT** (decimal value 2).
17	**n_numaux**	0, the number of auxiliary records for this entry.

6.8.6.3 The edata Symbol

The **edata** symbol marks the end of the **.data** section. Table 6-96 shows the format of the symbol table entry for **edata**.

Table 6-96. Format of the Symbol Table Entry for **edata**

Byte Offset	Field Name	Value
0-7	**_n**	The character string "edata" (padded to eight bytes with trailing zeros).
8-11	**n_value**	The virtual address of the end of the **.data** section.
12-13	**n_scnum**	-1
14-15	**n_type**	The enumerated constant **T_NULL** (decimal value 0).
16	**n_sclass**	The enumerated constant **C_EXT** (decimal value 2).
17	**n_numaux**	0, the number of auxiliary records for this entry.

6.8.6.4 The end Symbol

The **end** symbol marks the end of the **.bss** section. Table 6-97 shows the format of the symbol table entry for **end**.

Table 6-97. Format of the Symbol Table Entry for **end**

Byte Offset	Field Name	Value
0-7	**_n**	The character string "end" (padded to eight bytes with trailing zeros).
8-11	**n_value**	The virtual address of the end of the **.bss** section.
12-13	**n_scnum**	-1
14-15	**n_type**	The enumerated constant **T_NULL** (decimal value 0).
16	**n_sclass**	The enumerated constant **C_EXT** (decimal value 2).
17	**n_numaux**	0, the number of auxiliary records for this entry.

6.8.6.5 Other Symbols Defined in the AT&T COFF Template

The AT&T COFF template allows the symbol table to store entries for the following special symbols. Domain compilers, assemblers and linkers do not use these symbols.

.bb This entry marks the beginning of a block's symbol entries. All entries following this one, until the next **.eb** entry, describe the symbols found in a single block.

.eb This entry marks the end of a block's symbol entries.

.bf This entry marks the beginning of a function's symbol entries. All entries following this one, until the next **.ef** entry, describe the symbols found in a single function.

.ef This entry marks the end of a function's symbol entries.

.target This entry points to the structure or union returned by a function.

.*x*fake This entry marks the beginning of a structure, union, or enumeration that has no tag name. All entries following this one, until the next **.eos** entry, describe the symbols for elements within the structure, union, or enumeration.

.eos This entry marks the end of the symbol entries for a structure, union, or enumeration.

6.9 String Table

Following the symbol table is the COFF string table. You can find the string table by calculating where the symbol table ends, as follows:

$$strtab = symtab + SYMESZ * nument$$

where *strtab* is the byte offset from the beginning of the COFF file to the beginning of the string table.

symtab is the byte offset from the beginning of the COFF file to the beginning of the symbol table. This is the value of the **f_symptr** field in the file header.

SYMESZ is 18, the size of a symbol table entry in bytes.

nument is the number of entries in the symbol table. This is the value of the **f_nsyms** field in the file header.

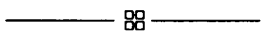
The COFF string table is a collection of null-terminated strings. It stores the strings referred to by any of the other COFF file parts, except the **.blocks** and **.symbols** sections. (The **.blocks** and **.symbols** sections, described in Sections 6.5.9 and 6.5.10, have their own string tables.)

> **NOTE:** In the AT&T COFF template, only the symbol table refers to strings in the string table. In Apollo COFF files, other parts of the file may also refer to the string table. For example, if a section name in an Apollo COFF file is longer than eight characters, the name is stored in the string table, and the section header's **_n** field refers to the string table. Keep this in mind when stripping the string table from a file.

The first four bytes of the COFF string table contain the size of the table, in bytes. The rest of the table is a contiguous series of null-terminated strings.

> **NOTE:** Offsets into the string table are from the beginning of the 4-byte size field, not from the beginning of the first string. The first string in the table, therefore, is at offset 4, not offset 0.

The **coffdump** or **dump** command with the **-c** option displays the string table.



Chapter 7

bind: The Aegis Linker

This chapter explains how to use the Aegis linking utility, **bind**. The object files that are output from **bind** may be executable programs, may be used as input to subsequent **bind** operations, or may be installed at run time as installed libraries.

The **bind** utility is similar in function to the BSD and SysV linking utility **ld** in that both join a number of object files into a larger object file.

> **NOTE:** At SR10, Apollo changed the format of object files. Before SR10, object files were in the **obj** format; and after SR10, they are in the COFF format. The SR10 version of **bind** may only be used to link COFF object files.

7.1 How to Invoke bind

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

$ bind *pathname...* [*option...*]

In other words, the command line simply consists of the word **bind**, one or more pathnames, and zero or more options. Note, also, that the command syntax shown above is somewhat misleading in that some options must precede pathnames but others must follow the pathnames. See Section 7.3 for information about specific options. You can use wildcards in pathnames.

The **bind** utility 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 librarians (**lbr** and **ar**) create valid library files.) The **bind** utility automatically loads all object modules stored in object files, but

conditionally loads the object modules stored in libraries. For details about how **bind** loads the object modules in library files, see Chapter 3.

Options, listed in Section 7.2, modify **bind**'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.

7.1.1 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
```

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 you are editing and recompiling.

7.1.2 Spreading a Binder Command over Several Lines

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

- Put **-end** at the end of the last 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
*
```


7.1.3 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 multi-line comment.

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

7.1.4 Errors

If a problem occurs during binding, the binder displays a message on 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.

Section 7.5 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, 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, the 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, the 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.
```

7.1.4.1 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, as long as the **-allocbss** option has not been used. See Section 7.1.1 for details.

7.2 bind 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:

-**b**inary *pathname*

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

-align *section_name* **long**
Aligns the named section on a 32-bit boundary at run time.

-align *section_name* **quad**
Aligns the named section on a 64-bit boundary at run time.

-align *section_name* **page**
Aligns the named section on a 8,192-bit boundary at run time.

-allkeepmark Preserves all marks.

-allmark Marks all global symbols in the input object files that appear after the option on the bind command line.

-allocbss Gathers all uninitialized global data from C programs, and allocates them all to a section named **.bss**.

-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 in *pathname* is to be "installed" when the output object file is invoked. (This is an alternative to using the **inlib** utility.)

-loadhigh Directs **bind** to write a record to the object file, which indicates that this object file should be loaded into the high end of a process's address space.

-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, and any other tools 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 Gathers all uninitialized global data and uninitialized data sections from C programs, and allocates them all to a 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.

-nlocalsearch, -nolocalsearch
Controls the order in which **bind** will search through library files to satisfy unresolved references.

-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.

-nomessages Suppresses informational messages at the end of a bind command.

-nomultires Omits error reporting for multiple 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.

-runtype *type* Specify the type of operating system environment (for example **sys5.3** or **bsd4.3**) that the program requires at run time.

-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)

-stacksize *number* Tells **bind** to write a record in the object file that specifies the size of the stack required by this program.

-system Do not make system globals visible.

-systype *type* Builds a shared resource record into the bound output module. Specify valid system names, such as **sys5.3**, or **bsd4.3**. 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 follow on the next line.

7.3 Detailed Descriptions of Each Binder Option

This section details each binder option.

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

FORMAT

-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 (8192-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 workstations, 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.

-allocbss	Allocates a section named **.bss** for uninitialized global variables generated by C.

FORMAT

-allocbss

DESCRIPTION

This option gathers all unallocated global variables from C programs and puts them in one section named **.bss**. If your program was compiled with **/bin/cc** or **/com/cc** with the **-bss** option, uninitialized global variables are not assigned to a section during compilation. A program is not executable in this state. You must use the **-allocbss** option on the final **bind** operation to join these uninitialized global variables into a section named **.bss**.

-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.

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:

```
# Without -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, or
/reiner/stand/by/me, or
~/truffaut/wild/child.bin, or
.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 this pathname, the binder reports a warning or error as appropriate.

If you specify the **-bdir** option, and the binder cannot find an input file with the pathname you gave, it goes on to search for that file in the directory named in the **-bdir** option. To do so, it appends the **-bdir** directory's pathname, a slash (/), and 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 directories, 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 constituent 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 directory 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 which object files will be in your working directory and which 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 creating.

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.

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

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 F816 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

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.) The **-include** option 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 4 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 by **bind**, but not by **lbr**. The specified pathname is used to locate the installed library object module.

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 information on installed libraries, see Chapter 4.)

At link time, **bind** checks the contents of libraries that will be installed at load time. If there is an external symbol that is undefined at link time, but is defined in an installed library, **bind** does not report this global.

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

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 the 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 this.

Instead of using multiple **-inlib** options on the same command line, you can 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, interlibrary 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 a 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 chapter 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 run time.

```
$ 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.
```

-loadhigh	Directs **bind** to write a record into the object file, which says that this object should be loaded into the high end of a process's address space.

FORMAT

-loadhigh

DESCRIPTION

The **-loadhigh** option causes **bind** to write a record to the object file, which directs the loader to put this file into the upper end of address space. Use this option when you are creating installed libraries. Installed libraries must be loaded into the upper end of address space, so another program can occupy the lower part.

-localsearch, -nolocalsearch, -nlocalsearch	Controls the order in which the binder will search through library files to satisfy unresolved external symbols.

FORMAT

-localsearch
-nolocalsearch | **-nlocalsearch** (these are synonyms)

DESCRIPTION

The **-localsearch** and **-nolocalsearch** options control the way the binder searches through a library file. Before reading this description, see Section 8.3, 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 7-1). (Since the desired version of **subcircle** precedes **circle** in the library, the binder will not load it.)

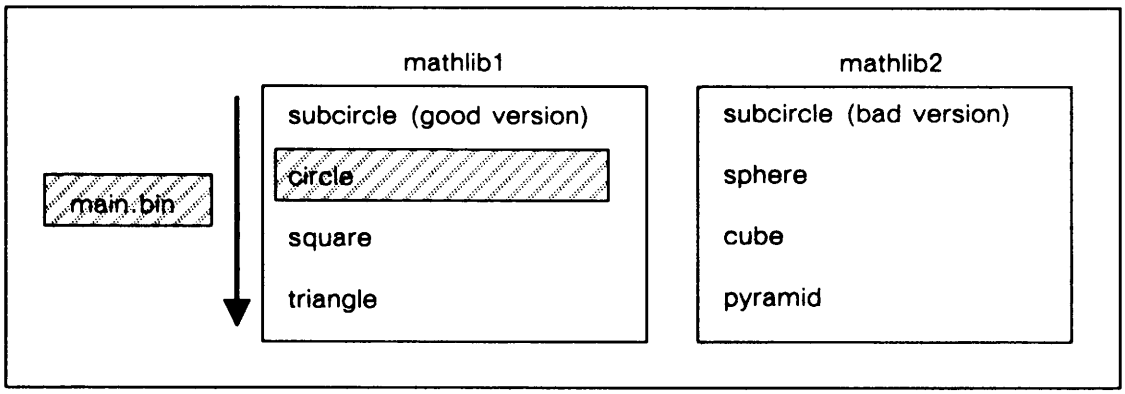


Figure 7-1. -nolocalsearch Option (Beginning of Search)

Next, the binder searches through **mathlib2** and therefore loads **subcircle** (as shown in Figure 7-2).

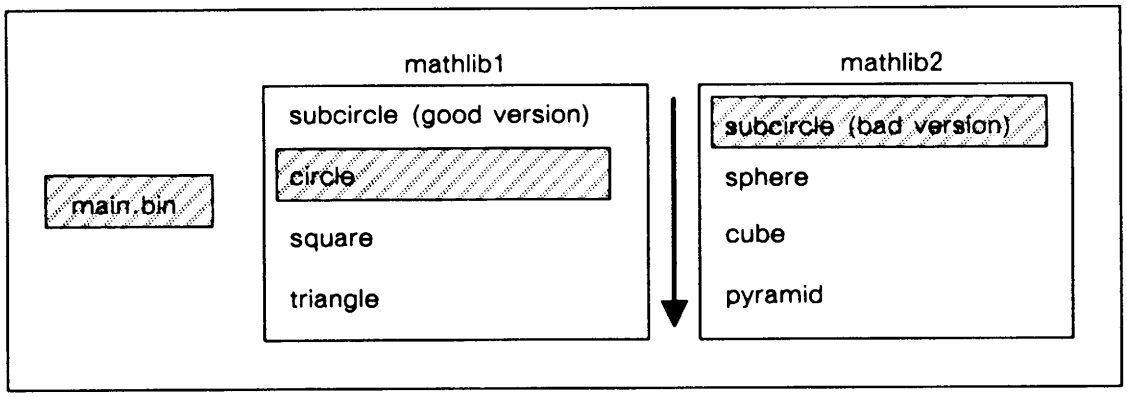


Figure 7-2. -nolocalsearch Option (End of Search)

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 7-1. 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 7-3 illustrates the search order.

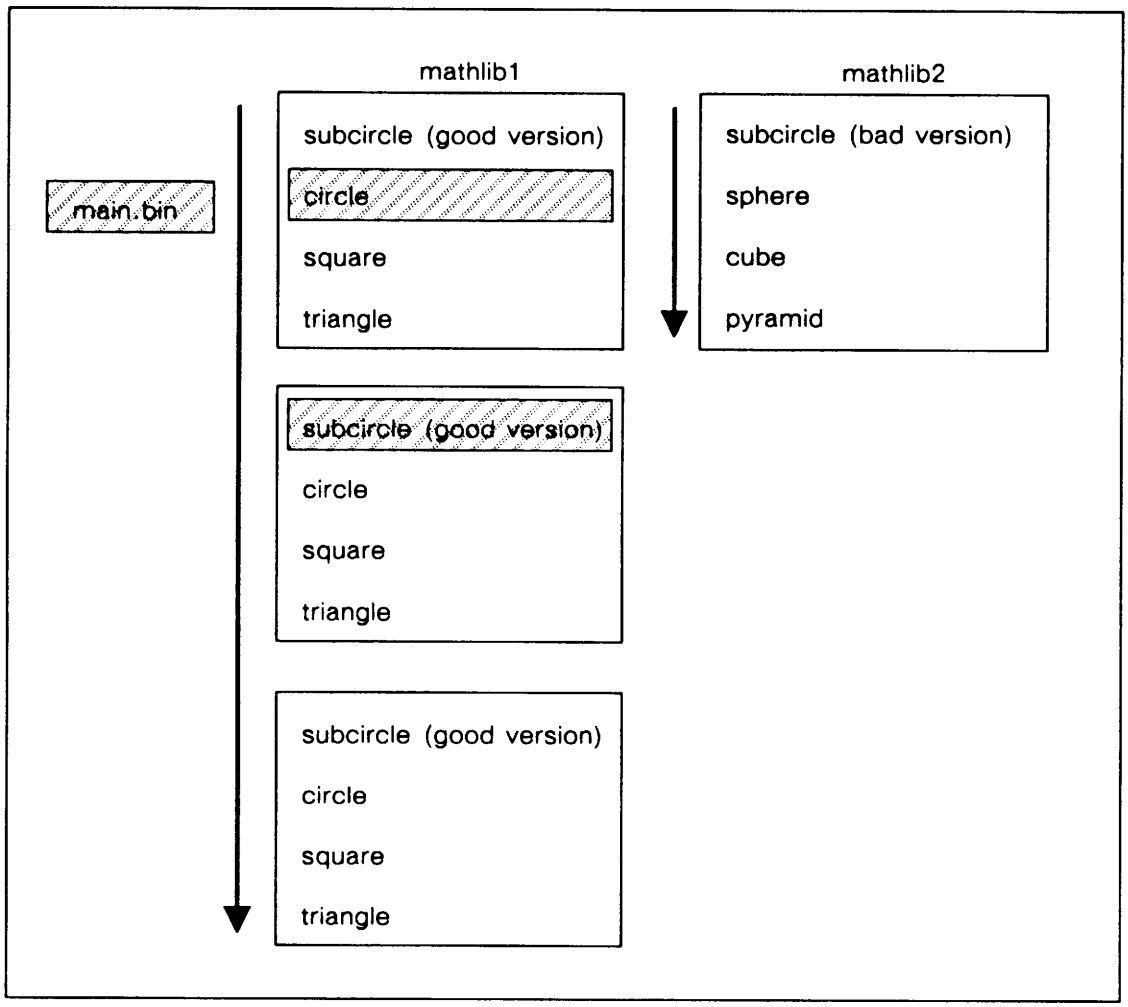


Figure 7-3. -localsearch Option (End of Search)

-looksection, -nolooksection, -marksection, -unmarksection	Controls the sharing of data sections between an executing object file and an installed library.

FORMAT

-looksection
-nolooksection
-marksection
-unmarksection

{ *section_name* | **-all** }

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. The **-all** option 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 run time 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 **-unmarksection** turn off the **looksection** and **marksection** attributes, respectively. 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 and compile with the **-nbss** option. (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

A P O L L O  Object Module Binder   6.05
1988/05/27 15:21:18 EDT (Fri)

File Name = abc
Module_Name = a_c  Version = 0.00
Absolute Code Program Module
Start Address = 00008260 = Offset 00000018 in Section 1

This object was made by the following:
   bind, Rev 6.05, Date: 1988/05/23 13:20:35 EDT (Mon)
   cc, Rev 5.05, Date: 1988/04/25 18:23:22 EDT (Mon)
```

```
Section Map:

 Id  Load Addr  Size     Name      Modules     Attributes

  1 00008248 00000048 .text                  R/O Concat Instr Long-aligned
            00000000 00000030 a_c
            00000030 00000010 b_c
            00000040 00000008 c_c
  2 00008290 00000070 .unwind                R/O Concat Data Long-aligned
  3 00010000 00000008 .aptv                  Concat Data Long-aligned
  4 00010008 00000000 .data                  Concat Data Zero Long-aligned
  5 00010008 00000004 big                    Ovly Data Zero Long-aligned
  6 0001000C 00000004 nick                   Ovly Data Zero Long-aligned
  7 00010010 00000004 str                    Ovly Data Zero Long-aligned
  8 00010014 00000004 rachel                 Ovly Data Zero Long-aligned
  9 00000000 0000001C .sri                   Info
 10 00000000 00000028 .mir                   Info
 11 00010018 00000000 .rwdi                  Concat Data
 12 00010018 0000003C .lines                 Debug Concat Data
 13 00010054 00000290 .blocks                Debug Concat Data

Global Map:

  Offset   In Section    Addr     Name

 00000018        1     00008260   <apollo_c_startup>
 00000030        1     00008278   b
 00000000        5     00010008   big
 00000040        1     00008288   catch
 00000000        1     00008248   main
 00000000        6     0001000C   nick        Marked
 00000000        8     00010014   rachel
 00000000        7     00010010   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   6.05
1988/05/27 15:21:18 EDT (Fri)
```

In this example, 6.05 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
Absolute Code Program Module
Start Address = 00008260 = Offset 00000018 in Section 1
```

a_c 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 **a_c** 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 run time. The start address in our sample is the instruction located 18 bytes past the beginning of section 1. 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**.

```
This object was made by the following:
   bind, Rev 6.05, Date: 1988/05/23 13:20:35 EDT (Mon)
   cc, Rev 5.05, Date: 1988/04/25 18:23:22 EDT (Mon)
```

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 manual 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** description in this manual.

The Global Map

Since the global map can also be generated by the **-globals** option, we describe this map in the **-globals** description in this manual.

-mark, -allmark, -allkeepmark, -unmark, -allunmark	Marks, unmarks, or preserves a mark on one or more global symbols.

FORMAT

-mark { *global_symbol* | **-all** }

-unmark { *global_symbol* | **-all** }

-allkeepmark

-allmark (a synonym for **-mark -all**)

-allunmark (a synonym for **-unmark -all**)

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 7.1.1 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 run time.
- If you install an object file produced by the binder, then the global symbols cannot resolve outstanding external references at run time 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 run time.}

... {run-time 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 run time.}

... {no run-time errors}
```

-mergebss	Combines uninitialized global variables and uninitialized data sections from C into one section named **.bss**

FORMAT

-mergebss

DESCRIPTION

Use this option to merge *uninitialized* C global variables and uninitialized data sections into a single section named **.bss**. Sections that correspond to *initialized* C variables do not get merged by the **-mergebss** option.

This option is provided for programmers who are using **/com/cc** without the **-bss** option, or are using **/bin/cc** with the **-nbss** option. In these cases, the C compiler creates a new section for each global variable. The section name in each case is the same as the name of the global variable. Use the **-mergebss** option to merge those sections that correspond to uninitialized C global variables into one **.bss** section. You can greatly reduce the number of sections, and in many cases, improve load performance by using this option.

> **NOTE:** The best way to decrease the number of sections in an object file is the use **/com/cc** with the **-bss** option, or to use **/bin/cc** without the **-nbss** option.

The C compiler gives the following attributes to the sections it creates for global variables: Ovly, Data, Zero, Long-aligned. (See Section 7.4 for more information about section attributes.)

Normally, the **-mergebss** option merges together all named sections that contain uninitialized data; however, this is not always the case. If a named section has a name that is visible to the 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 execution time.

When you use multilevel binding (see Section 7.1.1 for details) to develop a program, you should not use the **-mergebss** option until the final bind.

EXAMPLES

Consider the sample C source code files on the next page.

Contents of file "hi.c"

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

main()
{
    x = 2;
   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 these with the **-nbss** option. 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 creates a section named **.bss** which contains variable **x**. The variable **rachel** will remain in the section named **rachel** because it contains initialized data.

```
$ 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 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.

The **-module** option is particularly useful when you are preparing an object module to be passed on to **lbr**. Since **lbr** 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** *(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** suppresses 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 7.1.4).

-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 run time.
- In C, any global variable that was compiled with **-nbss** and that has not been allocated with the **-mergebss** option of **bind**.
- 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 (that is, 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
```

-runtype Indicates which UNIX environment this program must run under.

FORMAT

-runtype { bsd4.2 | bsd4.3 | sys5 | sys5.3 | any }

ARGUMENTS

bsd4.2	Program requires a 4.2BSD UNIX environment.
bsd4.3	Program requires a 4.3BSD UNIX environment.
sys5	Program requires a System V UNIX environment.
sys5.3	Program requires a System V Release 3 UNIX environment.
any	Program may run in any UNIX environment.

DESCRIPTION

An object module's **runtype** controls the system call semantics that the program requires. Normally, the **runtype** is taken from the object module's **systype**. Sometimes, however, you may want to develop a program that uses pathname resolution of one environment (say sys5), but uses system call semantics of another environment (say bsd4.3).

To do this, set the pathname environment with the **-systype** option and the different system call environment with the **-runtype** option. (See the **-systype** option for more information.)

> NOTE: Be especially careful about using the runtype **any**. Most programs are *not* independent of a particular operating system version.

EXAMPLE

You develop a BSD program, **prog1**, which you would like SysV users to use. It needs to obey BSD system call semantics, but SysV users will want pathnames resolved to /sys5. You could do this with the command:

```
$ bind  prog1.bin  -binary prog1  -systype any  -runtype bsd4.3
```

-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 00000048 .text                     R/O Concat Instr Long-aligned
              00000000 00000030 a_c
              00000030 00000010 b_c
              00000040 00000008 c_c
   2 00000070 .unwind                   R/O Concat Data Long-aligned
   3 00000000 .data                     Concat Data Zero Long-aligned
   4 00000004 big                       Ovly Data Zero Long-aligned
   5 00000000 .aptv                     Concat Data Long-aligned
   6 00000000 .rwdi                     Concat Data
   7 0000003C .lines                    Debug Concat Data
   8 00000290 .blocks                   Debug Concat Data
   9 00000000 .sri                      Info
  10 00000000 .mir                      Info
  11 00000004 nick                      Ovly Data Zero Long-aligned
  12 00000004 str                       Ovly Data Zero Long-aligned
  13 00000004 rachel                    Ovly Data 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 thirteen sections numbered 1 through 13.
- The total hexadecimal **Size** of each section—for example, the total size of section 2 is 70 bytes.
- The **Name** of each section—for example, .text, .unwind, big.
- 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:

  ```
  00000030 00000010 b_c
  ```

  It shows that object module **b_c** contributed 10 bytes of data to section 1, and that these bytes are offset 30 bytes from the start of section 1.
- The **Attributes** of each section — a set of attributes characterizes each section. For example, section 2 has the read-only, concatenate, data and long-aligned attributes.

-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.
```

-stacksize	Specifies the size of the stack required by this program.

FORMAT

-stacksize *decimal_number*

DESCRIPTION

-stacksize tells **bind** to write a record to the .sri section of the object file, stating the size of the stack that this program requires. The loader reads .sri records at load time and uses them to determine if the run-time environment is adequate for the program. When the loader sees a stack size record, the loader attempts to allocate the stated amount of stack space to the program.

-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 list the global symbols defined in installed libraries (including those specified by the **-inlib** option) when reporting "Undefined Globals." You can use this option to verify that the binder is, in fact, referring to the expected 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

$      -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	Indicates which UNIX environment this program is intended for.

FORMAT

-systype { bsd4.2 | bsd4.3 | sys5 | sys5.3 | any }

ARGUMENTS

bsd4.2	Program requires a 4.2BSD UNIX environment.
bsd4.3	Program requires a 4.3BSD UNIX environment.
sys5	Program requires a System V UNIX environment.
sys5.3	Program requires a System V Release 3 UNIX environment.
any	Program may run in any UNIX environment.

DESCRIPTION

An object module's **systype** controls the interpretation of pathnames. Different UNIX versions have different pathname formats. If your program uses one of the UNIX environments, you should specify which one to remove ambiguity.

By default, **bind** propagates the **systype** of input object modules by stamping the output object module with the same **systype**. The **-systype** option on **bind** overrides this default. The **bind** utility reports an error if input object modules are stamped with conflicting **systypes**. You must use the **-systype** option in this case to suppress the error the linker would otherwise report.

If you specify a **systype**, but not a **runtype**, the **runtype** is taken from the **systype**.

-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.

The **-xref** option 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.

EXAMPLE

```
$ bind -xref a.bin b.bin c.bin -binary my_program

All Globals are resolved.
Module Cross Reference

a_c  Compiled: 1988/05/27 14:47:02 EDT (Fri)

  Defined Globals:
    <apollo_c_startup>  big  main

  References To Globals:
    b

  References To Modules:
    b_c

  Sections:
    .aptv    .blocks  .data    .lines   .mir     .rwdi    .sri     .text
    .unwind  big
```

```
b_c     Compiled: 1988/05/27 14:47:04 EDT (Fri)
  Defined Globals:
    b     nick
  References To Globals:
    catch
  References To Modules:
    c_c
  Referenced By Modules:
    a_c
  Sections:
    .aptv    .blocks  .data    .lines   .mir     .rwdi    .sri     .text
    .unwind  nick

c_c     Compiled: 1988/05/27 14:47:05 EDT (Fri)
  Defined Globals:
    catch   rachel  str
  Referenced By Modules:
    b_c
  Sections:
    .blocks  .data    .lines   .mir     .rwdi    .sri     .text
    .unwind  rachel   str

   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
catch Defined In :c_c
  Referenced By Modules:
    b_c
main Defined In :a_c
nick Defined In :b_c
rachel Defined In :c_c
str Defined In :c_c
```

```
   Section Cross Reference

.aptv
  Defined In Modules:
    a_c  b_c
.blocks
  Defined In Modules:
    a_c  b_c  c_c
.data
  Defined In Modules:
    a_c  b_c  c_c
.lines
  Defined In Modules:
    a_c  b_c  c_c
.mir
  Defined In Modules:
    a_c  b_c  c_c
.rwdi
  Defined In Modules:
    a_c  b_c  c_c
.sri
  Defined In Modules:
    a_c  b_c  c_c
.text
  Defined In Modules:
    a_c  b_c  c_c
.unwind
  Defined In Modules:
    a_c  b_c  c_c
big
  Defined In Modules:
    a_c
nick
  Defined In Modules:
    b_c
rachel
  Defined In Modules:
    c_c
str
  Defined In Modules:
    c_c
```

7.4 Section Attributes

This section contains a list of the attributes that can characterize a section. This list should help you interpret the maps produced by the the **bind -map** option, or the **lbr -list** option. The boldfaced portion of the attribute is the abbreviation that you see in the **lbr** and bind maps. For example, the expression "**abs**" appearing in a binder map refers to the **abs**olute attribute.

compress attribute

This flag indicates that the section is stored in the object file in a compressed form. The section will be expanded at load time by the loader, using information stored in the COFF object file.

data and **instr**uction attributes

A section may have either the data attribute or the instruction attribute. The instruction attribute means that the section contains machine code instructions only. The data attribute means the section is not an instruction, information or library section. Users cannot control these attributes.

debug attribute

bind reports this attribute if the STYP_DEBUG flag is set in the COFF section header.

info attribute

bind reports this attribute if the STYP_INFO flag is set in the COFF section header.

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 Section 7.3.

lib attribute

bind reports this attribute if the STYP_LIB flag is set in the COFF section header.

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 1024 bytes. You can control these attributes through the **-align** binder option described in Section 7.3.

look_installed (or **looksection**) attribute

A section may or may not have the **look_installed** attribute (also called the **looksection** attribute). At run time, 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 Section 7.3.

overlay and **concate**nated attributes

A section, that is not an info or lib 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 run time. Instead, they are placed one after another.

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 run time 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 Section 7.3.

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 run time. 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.

7.5 bind Error and Warning Messages

This section 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 Chapter 3 for details on start addresses.)

Bad section number in symbol table entry (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.

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 target machines (error)

The "magic numbers" in the object files (which specify the target machines) don't match.

Conflicting object SYSTYPE 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** description earlier in this chapter.

Conflicting object RUNTYPE system types (error)

The binder prohibits you from specifying input object files having different runtypes. For example, suppose that the compiler stamped **a.bin** with a runtype of **sys5**, but **b.bin** has a runtype 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 runtype.

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.

Global already entered into global table (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.

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.

Improperly terminated string table entry (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.

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.

Installed library pathname length exceeds maximum (error)

The pathname of the installed library that you specified is too long.

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** or **-runtype** option was not a valid name or was entered incorrectly. Enter a valid system name. Also, correct any format or typographic errors.

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** option.

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.

No symbol table (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.

No symbol table entry for global (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.

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.

Not an object module or a library (error)

The binder was expecting a file containing either a COFF object module or a COFF library file. The file you specified was neither. Perhaps you tried to bind an object file in **obj** format.

Object module contains no relocation info (error)

This is probably the result of using the **-a** option on **ld**, or stripping the relocation information with the UNIX **strip** command.

Object module has been stripped (error)

STRIPPED flag is set in the COFF file header. File has been stripped with the UNIX **strip** command or the **-s** option on **ld**.

Out-of-bounds string table offset (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.

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.

R/O section refers to installed library symbol; may need compile-time -inlib (warning)

An external symbol that is referenced in the .text section was not seen at compile time in the Known Global Table. As a result, the compiler created a direct reference to the symbol, assuming it would be bound in. But, the symbol is actually in an installed library, so the reference to it should be through the transfer vector. You should go back and recompile using the **-inlib** compile option, and list the library with that option.

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 3072 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.

There are unallocated .bss globals (warning)

You should bind again with the **-allocbss** option.

Too many sections in input file (error)

One of your input object modules has more than 3072 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.

Unknown relocation type (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.

Virtual address not within bounds of any section (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.

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.

Chapter 8
lbr: The Aegis Librarian

You use the Aegis librarian, **lbr**, to create, edit, or describe a **library file**. A library file consists of one or more object modules collected together for easy access by one of the linkers (**bind** and **ld**).

Before working with **lbr**, there are several things you should know about the Domain/OS programming environments and library files:

- The SysV and BSD environments of Domain/OS contain the UNIX librarian **ar**, which is described in the *SysV Command Reference* and the *BSD Command Reference*.
- Beginning with SR10, **lbr** produces library files that are compatible with those created by standard UNIX **ar**.
- Beginning with SR10 of Domain/OS, **lbr** will only accept object files in COFF format.

This chapter details the following topics:

- How to invoke **lbr**.
- How to create a library file.
- How **lbr** analyzes command lines.
- How to spread an **lbr** command over multiple lines.
- How to embed comments in an **lbr** command.
- How the linkers analyze library files.
- How **bind** sometimes use library files to determine program start address.

- How to use all the **lbr** options.
- The error and warning messages you can get from **lbr**.

8.1 Invoking lbr

Use one of the following formats to invoke **lbr**:

$ lbr **-create** *library_pathname object...* [*option...*]

$ lbr **-update** *library_pathname* [*object...*] [*option...*]

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 (that is, a file produced by either the compiler or the linkers) 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 **lbr** 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 **lbr** to write or suppress purely informational messages.
-quit	Causes **lbr** 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.

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, **lbr** issues an error message.

8.1.1 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
```

8.1.2 Order of Execution

lbr 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 cos.bin -list
```

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

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

If your command line contains an error, **lbr** 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) does not contain a valid object module. In this case, **lbr** 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.

8.1.3 Spreading lbr Commands over Several Lines

If you want to spread an **lbr** 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 **lbr** command, you must either put **-end** at the end of the command or leave the final line blank. For example, the following three **lbr** 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
$
```

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

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

8.1.4 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
```

8.2 Error and Warnings

If **lbr** detects a problem with the command line, it issues either an error message or a warning message. An error message indicates that **lbr** could not perform the requested operation or that some error condition arose while **lbr** 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:

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

Section 8.6 contains a complete list of all **lbr** error and warning messages, and an explanation of the likely cause of the problem.

8.3 How the Linkers Scan Library Files

Here's how **bind** scans to resolve external symbols: **bind** starts scanning at the first object file or library file on the command line and moves to the right. As it scans, it automatically loads all object modules stored in object files and all **-included** object modules from libraries. Any of these object modules may contain external references and definitions. The **bind** utility first tries to resolve these external references and external definitions in object modules already loaded. When **bind** 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.

If there are some unresolved external references, **bind** rescans. On a rescan, **bind** only searches library files, and searches them in the order you entered them on the command line. On a rescan, **bind** attempts to satisfy outstanding unresolved references by using unloaded modules from library files. This process continues until **bind** determines either that all external references are resolved or that no further resolutions can be made. **bind** always scans libraries in the order presented on the command line. Within a library, **bind** scans libraries in the order that they appear in the report generated by the **-list** option of **lbr**.

Here's how **ld** scans to resolve external symbols: **ld** starts scanning at the first object file or library file on the command line and moves to the right. When it is scanning an object file, **ld** automatically binds the object module stored in the object file. If the file it is scanning is a library, **ld** scans that library again and again, as long as **ld** is finding symbols it needs in the library. When **ld** finishes scanning a library, it does not go back to it. So, the order of libraries on an **ld** command line is important.

8.4 Program Start Address

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

The linker 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 address, you can use the **-entry** option on **bind** or the **-e** option on **ld** to define a nondefault one.)

The **bind** utility 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 **bind** sets the start address to the first possible start address it encounters.

- If no input object module defines a possible start address, then **bind** looks for a possible start address in the unloaded library object modules. **bind** makes this search only if it would have to scan the libraries anyway to satisfy unresolved external references. That is, on **bind**'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 **bind** (concurrent with its search to satisfy unresolved external references) searches the unloaded library object modules for a possible start address. **bind** will load the first library object module that defines a possible start address (even if it does not satisfy an outstanding external reference). If **bind** 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.

8.5 Detailed Descriptions of Each lbr Option

This section is devoted to detailed descriptions of each **lbr** 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, **lbr** issues a warning. Note that **lbr** 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 **lbr** is case-sensitive to *object_module_name*.

-o *pathname*

The **-o** *pathname* is optional. If you specify it, **lbr** copies the object module into pathname. If you do not specify **-o** *pathname*, **lbr** 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.

EXAMPLE

```
$ lbr -upd mylbr -list

rwxrwxrwx 12648/    12   1064 Jun  3 10:16 1988 circle.bin (circle)
rwxrwxrwx 12648/    12   1064 Jun  3 10:16 1988 square.bin (square)
```

-messages, -nomessages	Reports or suppresses a report on the number of errors and warnings encountered in an **lbr** 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 **lbr** to either report or suppress a summary of the number of errors and warnings that occurred in a **lbr** session. **-messages** (the default) forces the report, and **-nomessages** suppresses the report.

-messages also tells **lbr** to produce informational messages during successful execution.

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.
```

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.
```

-quit	Causes **lbr** to ignore everything that appears after this option on the command line.

FORMAT

-quit

DESCRIPTION

The **-quit** option causes **lbr** to ignore everything that appears after it on the command line. The **lbr** utility 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 **lbr** 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 **lbr** utility handles the **-replace** *pathname* option in the following way.

1. **lbr** reads the file stored in *pathname* to learn which object module(s) is stored there.
2. **lbr** 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, **lbr** 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, **lbr** 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
```

8.6 lbr Error and Warning Messages

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

An **error message** indicates that **lbr** could not perform the requested operation or that some error condition arose while **lbr** 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:

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

Following is a list of all the error and warning messages produced by **lbr**:

Cannot open file (warning)

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

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)

lbr interprets the first character string after **-create** as the filename of the new library. **lbr** 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 in archive format (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 **lbr** or **ar**. The **lbr** utility will not alter the specified file.

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.

Is not a COFF object (error)

You tried to add a file that is not in COFF format to the library.

Module not found (warning)

When you used the **-delete** option, you specified an object module that was not part of the library file. (Note that **lbr** 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 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, **lbr** 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, **lbr** 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.

-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 **lbr**. If you enter **-create** twice, **lbr** 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. **lbr** 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. An **lbr** command cannot contain both **-create** and **-update**. If this is the only error, then **lbr** 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 **lbr** 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. (**lbr** assumes that arguments beginning with a hyphen are options.) **lbr** 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.

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 **lbr** option. Perhaps you misspelled an **lbr** option, or perhaps you accidentally put a hyphen in front of a filename. To let you know where you went wrong, **lbr** 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)

lbr interprets the first character string after **-update** as the filename of an existing library. **lbr** 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.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Reader's Response

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

Document Title: *Domain/OS Programming Environment Reference*
Order No.: 011010-A01

User Profile

Your Name _____ Title _____

Company _____

Address _____

Telephone number (____) _____

When you use the Apollo system, what **job(s)** do you perform?

☐ Programming ☐ Application End User ☐ Hardware Engineering
☐ System administration ☐ Other (describe) _____

Characterize your level of **experience** in using the Apollo system:

☐ Experienced user (2+ yrs.) ☐ New user (6 mos. or less)
☐ Moderately experienced user (6 mos.–2 yrs.)

What programming **languages** do you use with the Apollo system?

_____

Distribution

How do you know what manuals are **available** to support the products you're using or want to use?

_____

What is a major concern for you in **ordering** books?

_____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: _____

_____

_____

_____

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 MAIL 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

cut

Reader's Response

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

Document Title: *Domain/OS Programming Environment Reference*
Order No.: 011010-A01

User Profile

Your Name _____ Title _____

Company _____

Address _____

Telephone number (____) _____

When you use the Apollo system, what **job(s)** do you perform?

☐ Programming ☐ Application End User ☐ Hardware Engineering
☐ System administration ☐ Other (describe) _____

Characterize your level of **experience** in using the Apollo system:

☐ Experienced user (2+ yrs.) ☐ New user (6 mos. or less)
☐ Moderately experienced user (6 mos.–2 yrs.)

What programming **languages** do you use with the Apollo system?

_____

Distribution

How do you know what manuals are **available** to support the products you're using or want to use?

_____

What is a major concern for you in **ordering** books?

_____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: _____

_____

_____

_____

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 MAIL 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

cut

Reader's Response

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

Document Title: *Domain/OS Programming Environment Reference*
Order No.: 011010-A01

User Profile

Your Name _____ Title _____

Company _____

Address _____

Telephone number (____) _____

When you use the Apollo system, what **job(s)** do you perform?

☐ Programming ☐ Application End User ☐ Hardware Engineering
☐ System administration ☐ Other (describe) _____

Characterize your level of **experience** in using the Apollo system:

☐ Experienced user (2+ yrs.) ☐ New user (6 mos. or less)
☐ Moderately experienced user (6 mos.-2 yrs.)

What programming **languages** do you use with the Apollo system?

_____

Distribution

How do you know what manuals are **available** to support the products you're using or want to use?

_____

What is a major concern for you in **ordering** books?

_____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: _____

_____

_____

_____

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 MAIL 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

cut