DOMAIN Assembler Reference

Order No. 008862
Revision 01

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: January, 1986
Latest Printing: January, 1987

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

Preface

The *DOMAIN Assembler Reference* describes the assembly language used in DOMAIN systems. The purpose of the manual is to provide the reader with information about writing and debugging DOMAIN assembly language programs, calling assembly language routines from a high-level language, and interpreting object module format in order to write a compiler compatible with the DOMAIN system.

The intended reader is an experienced assembly language programmer who is also familiar with programming in FORTRAN, Pascal, or C on the DOMAIN system. The reader should have a solid understanding of one or more of the following microprocessors: MC68000, MC68010, MC68020. Also, because DOMAIN assembly language uses most of the 68000 series instruction set, the reader should have appropriate Motorola documentation to use as a companion to this manual.

We've divided the manual into two parts. Part One contains DOMAIN assembly language reference material; Part Two provides useful information about DOMAIN run-time conventions. The Appendixes provide additional information about the DOMAIN assembler, the low-level debuggers, SR9.0 calling conventions, and the object module format. Each part covers a specific type of information and builds on the preceding part. However, depending on the reader's familiarity with the DOMAIN system, the parts may be read out of sequence.

We've organized this manual as follows:

PART ONE	Provides fundamental information about DOMAIN assembly language, such as address modes, source file format, and naming conventions. This part assumes that the reader is familiar with one or more Motorola assemblers, but may not be familiar with DOMAIN's assembly language implementation.
Chapter 1	Provides a conceptual overview of DOMAIN internals and provides the background information needed to use DOMAIN assembly language.
Chapter 2	Illustrates how ASM (the DOMAIN assembler) works, and describes how to invoke the assembler.
Chapter 3	Introduces various aspects of the DOMAIN assembly language. The first half of the chapter illustrates the source file and discusses the elements of the language, such as source file formats, labels and special characters, format of numbers, strings, and register lists, and instruction formats. The first half also provides an overview of instruction conventions, pseudo-ops, and directives. In addition, this chapter describes the four types of operators and details naming conventions The second half of the chapter provides a brief definition and an example of each addressing mode.

Chapter 4	Describes DOMAIN assembly language pseudo-ops and directives. Details each pseudo-op and directive, and provides examples.
Chapter 5	Describes the listing file and related topics, such as special symbols and the cross-reference listing. Provides a sample listing as an example.
PART TWO	Presents the run-time conventions of the DOMAIN system. This part assumes that the reader has knowledge of compilers and understands the concepts of calling conventions and floating-point numbers, but needs information specific to the DOMAIN system.
Chapter 6	Discusses calling conventions topics, such as the stack, prologue and epilogue code, and stack unwinders. Provides many examples of how to call a DOMAIN assembly language routine and how to pass parameters from DOMAIN assembly language.
Chapter 7	Provides information about two mathematical libraries. The first library is the Integer Arithmetic Library, which implements 32-bit operations not supported by the processor hardware. The second library is FPP, the Floating-Point Package. The chapter discusses FPP implementations, calling and exiting conventions, and gives a list of FPP operators.
APPENDIXES	Provides specific and detailed information on both DOMAIN assembly language and DOMAIN system topics.
Appendix A	Explains error codes and messages.
Appendix B	Provides a list of DOMAIN assembly language's legal instructions, valid machine types, and legal suffixes.
Appendix C	Lists the TERN instruction set.
Appendix D	Discusses how to use the low-level debuggers DB and MDB, and provides a list of DB and MDB commands.
Appendix E	Discusses pre-SR9.5 calling conventions, ECB (Entry Control Block), the stack, and prologue and epilogue code. Provides examples of how to call a DOMAIN assembly language routine and how to pass parameters from ASM.
Appendix F	Discusses the object module in two parts: overview and application. The overview provides a theory of operation, which includes how ASM generates an object module, the role of the binder and loader, and how to display an object module listing. The application part uses a sample object module to illustrate the object module format, which is discussed in detail.

Summary of Technical Changes

To improve overall performance, SR9.5 makes changes to the runtime environment. In general, these changes do *not* affect user programs, since most programs do not depend on internal information regarding the runtime environment. However, OEMs, software suppliers, and anyone coding in assembly language, could depend on details within the runtime environment.

Specifically, you need to adjust your programs if you depend on the following internal structures:

More Registers Saved	DOMAIN compilers preserve more registers across routine calls: registers A2–A4, D2–D7, and FP2–FP7. ASM routines using registers must be adjusted because SR9.5 register saving conventions require routines to preserve more registers.
Stack Frame Format	The stack frame format now saves more registers, and eliminates the pointer to the ECB block, unit word, and optional DB field. There are now two formats: one for routines that save FP registers, and one for those that do not. ASM routines that depend on pre–SR9.5 stack frame format must be adjusted because the SR9.5 stack frame eliminates fields containing the entry control block (ECB) pointer, unit word, and optional saved DB, and adds a new format because of new register saving conventions. ASM routines that depend on the entry control block (ECB) structure must change.
Prologue and Epilogue Code	Assembly language routines' prologue and epilogue code have been revised to support changes to register saving conventions and stack frame format.
Targets of Stack Unwinders	Routines that are targets of stack unwinders (such as the caller of **pfm_$cleanup**) must now preserve all registers. ASM routines that call **pfm_$cleanup** must preserve A and D registers because **pfm_$cleanup** does not restore them.
Symbolic Tracebacks	Assembly language routines require a **LINK** instruction so that the symbolic traceback mechanism (such as the **traceback (tb)** command) can list the routine's name in a traceback. Without a **LINK** instruction, the call of the routine will *not* be listed. ASM routines should use a **LINK** instruction in their prologue code to provide adequate information to tools performing symbolic tracebacks such as **tb (traceback)**, **debug** and **dpat**.

Procedure Pointers

Procedure pointers now point to the address of the routine, rather than to a data structure containing the ECB address. This affects you if you access the Known Global Table (KGT), install your own global libraries, and write type managers in C using the Open Systems Toolkit.

Any routines that depend on the structure of procedure pointers must be adjusted because SR9.5 uses a new procedure pointer format.

Related Manuals

For more information on topics related to DOMAIN assembly language, see the appropriate Motorola manuals and the following DOMAIN system manuals:

- *AEGIS Internals and Data Structures* (N/A - available to OEMs)
- *DOMAIN Binder and Librarian Reference* (004977)
- *Programming With General System Calls* (005506)
- *DOMAIN C Language Reference* (002093)
- *DOMAIN FORTRAN Language Reference* (000530)
- *DOMAIN Pascal Language Reference* (000792)

Problems, Questions, and Suggestions

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

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

```
$ help crucr <return>
```

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

Documentation Conventions

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

UPPERCASE	Bold, uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.
lowercase	Bold, lowercase words or characters in formats and command descriptions represent values that you must supply.
bold	Bold words in text introduce a new term.
`input/output`	Typewriter font words in command examples represent input or literal system output.
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions. In sample Pascal statements, braces assume their Pascal meanings.
|	A vertical bar separates items in a list of choices. Combined with angle bracket, they indicate an optional item.
< >	Angle brackets enclose the name of a field in which you supply information.
...	Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.
.	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.
.	
.	

Contents

Illustrations

Tables

PART 1: DOMAIN Assembler Reference

Chapter 1

Introduction to DOMAIN Assembly Language

This introductory chapter presents and defines some basic concepts that you will use throughout the manual. While some of the information contained within this section is specific to the assembler, many of the concepts are related to DOMAIN system architecture, or the DOMAIN system. The topics we discuss are:

- What is DOMAIN assembly language?
- Overview of the DOMAIN program environment
- Run-time and calling conventions

1.1 What is DOMAIN Assembly Language?

The DOMAIN assembly language is the assembly language for DOMAIN processors. It assembles instructions and data for the following Motorola MC68000-family processors into DOMAIN format object modules:

- MC68000 CPU
- MC68010 CPU
- MC68020 CPU
- MC68881 Floating-Point (FP) Coprocessor
- MC68851 Memory Management Unit
- TERN CPU

NOTE: The TERN CPU is the propriety processor used in the DN460 and the DN660 workstation models. It supports both the MC68010 instruction set and a floating-point instruction set that is similar to the MC68881.

1.1.1 What You Need to Know

To use the assembler, you first must understand

- The architecture and instruction set of the processor(s) you are programming.
- The representation of instructions and data in assembly language, and the procedures for using the assembler.
- The environment in which your program executes, in particular the run-time and calling conventions required for your program to interact with compiled routines and the operating system.

This manual does not discuss the architecture and instruction set of the processor(s) you are programming (first topic). We assume that you are familiar with 68000-family processors and have the appropriate Motorola documentation. Refer to the preface for more information.

The manual discusses the second and third topics. Although assembly language reflects some of the run-time conventions, you need to understand the assembly language examples. Therefore, the remainder of this chapter presents an overview of the DOMAIN program environment, which should help you understand the *environment-dependent* aspects of the DOMAIN assembly language. Part One of the manual discusses the language in detail; Part Two describes the run-time conventions and mathematical libraries in full.

We assume that you are familiar with aspects of the DOMAIN system such as high-level programming and the process of translating, binding, and executing a program. Additionally, you should be generally familiar with the system call mechanism, as described in *Programming with General System Calls*.

1.2 Overview of the DOMAIN Program Environment

This section presents the background you need to understand the DOMAIN run-time environment and calling conventions, such as:

- Address space
- Mapping
- Object modules
- Installed libraries

The last section of this chapter summarizes the actual run-time environment and calling conventions.

1.2.1 Address Space

Figure 1–1 illustrates the structure of the virtual address space in which a process runs. The specific addresses in the illustration are only examples; the actual size and partitioning of the address space varies with machine type and may change between software releases.

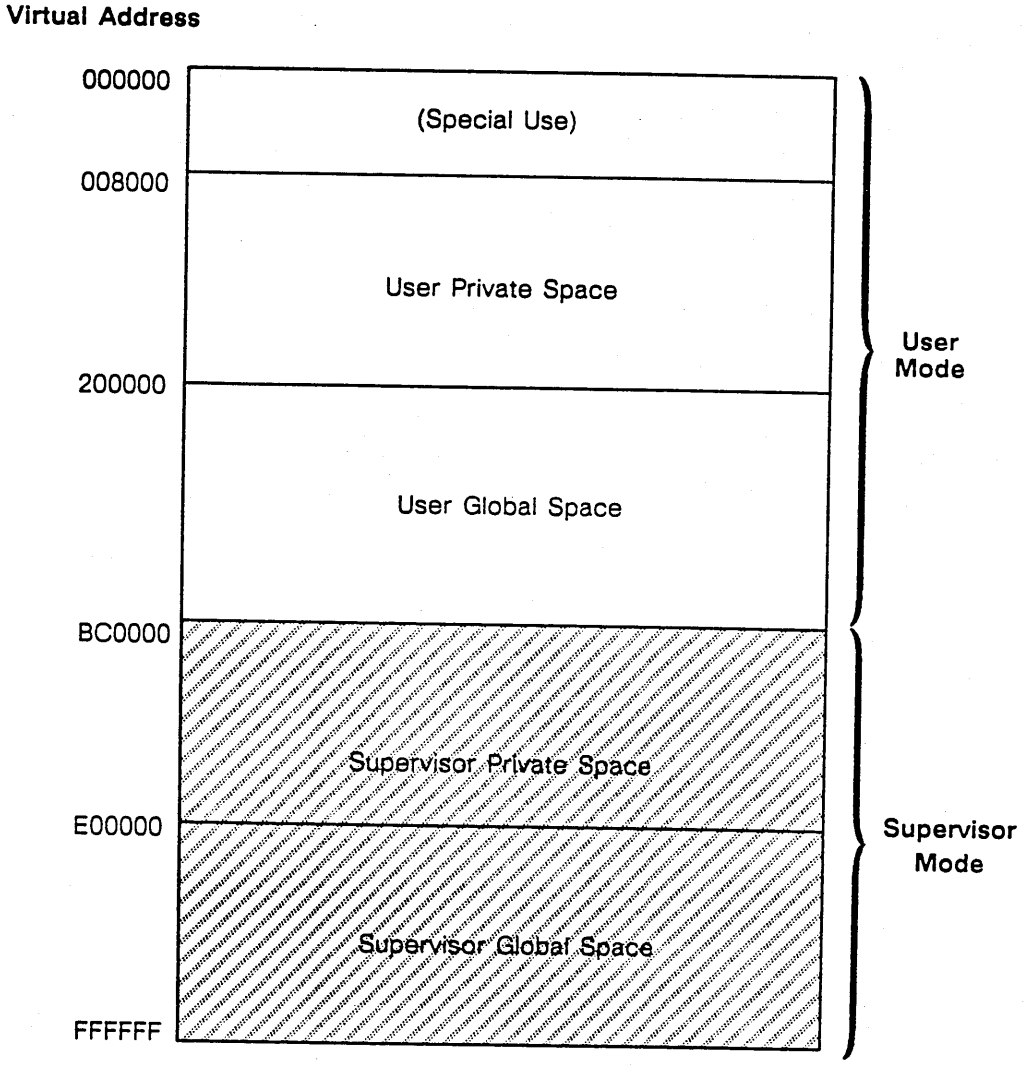


Figure 1–1. Process Address Space (illustrated for 16 Mb virtual address space)

User private space is accessible to only one process. User programs and data are normally loaded and run in this space.

User global space is shared by all processes running on the node. Two processes that refer to the same address in this range access the same physical byte. User global space is primarily occupied by **shared libraries**, such as the run-time libraries for the various languages. For more information, refer to the section "Installed Libraries" within this chapter.

Supervisor space is occupied by the operating system kernel and data that it controls. Like user space, supervisor space is divided into global and private areas. However, supervisor space is not directly accessible to programs running in user mode.

1.2.2 Mapping

Storage management in the DOMAIN operating system is based on a **single-level store** model in which no distinction is made between primary (main memory) and secondary (disk) storage. Processes access files, or any objects, by **mapping** the files into their address space and operating on them using ordinary machine instructions. The virtual memory management system pages data between files (local or remote) and physical memory on demand.

For example, suppose a process maps the file `BETH.DAT` into virtual addresses 500000-501000. No data transfer occurs as a result of the mapping. To inspect the first byte of the file, the process simply references the byte at 500000 (perhaps using a **MOVE** byte instruction). The first time this happens a page fault occurs and the virtual memory manager services the fault by copying the first page (1024 bytes) of the file from the disk or over the network to an available page frame in physical memory. If the process writes into the mapped file, the memory manager ensures that the changed pages eventually are written back to the disk.

Mapped files are the only storage available to processes. Physical memory serves only as a cache over files. Storage that might normally be thought of as simply *memory*, such as the stack or dynamically allocated memory, is backed up by a temporary file known as the **stack file**, which is created for each process. There is no separate disk swapping area for virtual memory management.

Conversely, mapping is the only way a process can access a file. While there is no explicit data transfer between disk and main memory, the streams facility simulates a more traditional I/O interface on top of the mapping mechanism. The streams facility hides the details of mapping and presents a familiar read/write style interface. Thus, most user programs use the streams interface, or higher-level calls based on it, to do I/O. Nevertheless, mapping underlies all file I/O operations and is also directly available to any program through system calls.

Two or more processes can map the same file simultaneously. If the processes are on the same workstation, they share the same physical data bytes, whether they are on disk or cached in memory. This provides a shared memory facility that, for example, is the basis of the mailbox interprocess communication mechanism. Processes on different workstations necessarily have separate physical memory caches; to avoid consistency problems, these processes are restricted to read-only access to shared files. Note that this is the only logical distinction between local and remote file access in the DOMAIN file system. Concurrent file access is subject to locking restrictions, which the programmer specifies.

1.2.3 Object Programs

Mapped files allow rapid loading and automatic sharing of DOMAIN object modules. An object program (represented by a `.bin` file name extension) consists of a set of independently loadable **sections**. The two principal types of sections are

- **Pure sections** — contain read-only code and constant data. Pure sections are stored in the object module in memory image format. When the module is loaded, the sections are mapped into the address space of the process. The contents are then paged on demand into physical memory directly from the object module file. Pure sections are fully shareable among multiple processes running the same program.

- **Impure sections** — contain writeable data and any necessary absolute address constants. The object module contains a template, which describes the size and required initialization for each impure section. The loader allocates free space for the section and initializes it as directed by the template. Each loaded instance of a program has a separate copy of its impure sections.

By default, most programs have these three sections:

- **PROCEDURE$** — a pure section that contains code and constant data.

- **DATA$** — an impure section that contains static data and address constants.

- **DEBUG$** — a pure section that contains debugging information.

The programmer can define additional or alternate sections. When object modules are bound together, sections with the same name are concatenated or overlayed, depending on information in the object module.

DOMAIN programs are **position independent**, which means that they can be loaded and run anywhere in the address space of a process. References to external routines and data are resolved to actual addresses when the program is loaded.

The loader is unable to insert addresses into pure code sections, since the sections are strictly read-only. Therefore, pure code cannot contain any absolute addresses. Access to external routines and data is done indirectly through address constants stored in an impure section.

1.2.4 Installed Libraries

The DOMAIN operating system provides two ways to make procedures in one object module available to another module. Consider an object program that calls procedures in another *library* object module:

- The two modules can be *bound* together to produce a new object module that contains both sets of code, with the references between them resolved.
- The library can be *installed* in the address space of a process, making it resident in virtual memory for the life of the process.

When an object module is installed, its entry point names and their associated addresses are recorded in a process data structure called the **Known Global Table (KGT)**. The loader uses the KGT to resolve references to installed library routines when a program is loaded for execution.

Most DOMAIN system libraries, such as run-time language support, streams, graphics, etc., are installed in the global portion of address space where they are accessible to all processes. Thus, object modules produced by compilers can often be executed directly, without a separate linkage step to bind library routines.

You can install your own libraries, either in the global or private address space. Note that installing a library is primarily a mapping operation; that is, code does not physically move from the disk until it is referenced and brought in by a page fault. An installed routine that is never called uses only virtual address space and takes no other resources.

1.3 Overview of Run-Time and Calling Conventions

In the DOMAIN program environment conventions, three of the 68000 address registers have special functions:

- **A7** — the Stack Pointer (**SP**) points to the top of the call stack. The stack grows from high addresses to low addresses.
- **A6** — the Stack Base (**SB**) points to a fixed position in the stack frame of the currently active routine. Local variables and arguments are accessed relative to this register.
- **A5** — the Data Base (**DB**) points to the start of the impure data section (usually **DATA$**) associated with the active routine. Static data is accessed relative to DB.

Figure 1-2 illustrates the stack frame of a typical procedure. Following the illustration is the sequence of events in a procedure call.

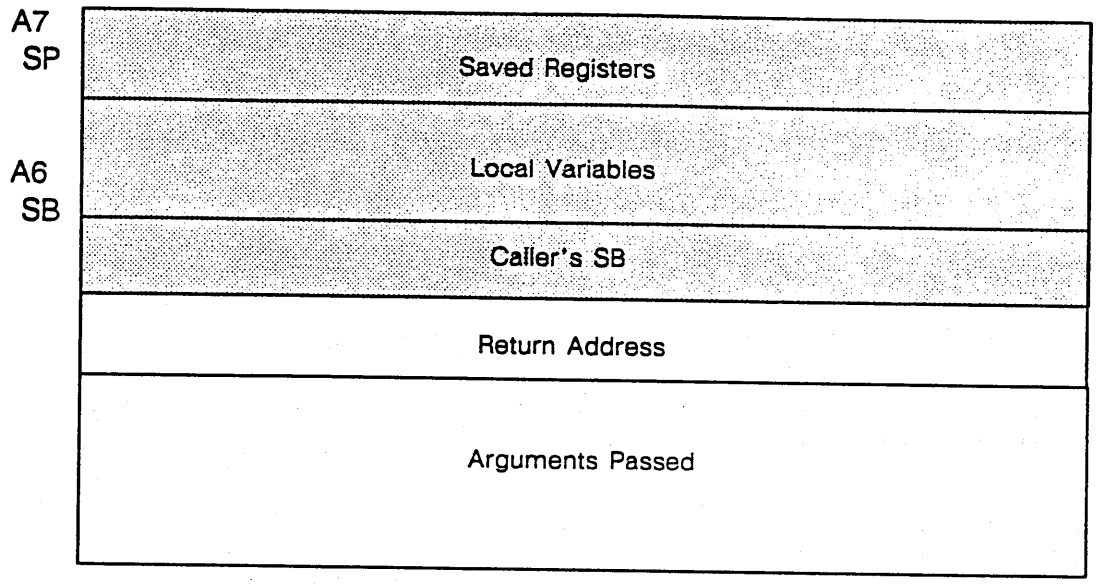


Figure 1-2. Procedure Stack Frame

NOTE: Shaded area indicates callee's responsibilities; unshaded area indicates caller's responsibilities.

1. Caller pushes arguments onto the stack in reverse order so that the first argument is at the top of the stack.
2. Caller executes a **JSR** or **BSR** instruction to push the return address onto the stack and transfer control to the procedure.
3. Called procedure executes a **LINK** instruction to establish the SB register value and allocate stack space for local variables. **LINK** also provides a thread of pointers linking successive call frames.
4. Callee pushes the registers it can change onto the stack. The callee is responsible for preserving most of the caller's registers.
5. Callee executes.
6. Callee restores the caller's registers from the stack. If it is a function, the callee can leave a return value in D0 or A0.
7. Callee executes the **UNLK** instruction to pop local data off the stack and restore the caller's SB register.
8. Callee returns to the caller using the **RTS** instruction.
9. Caller pops arguments off the stack.

As you will recall from the section "Object Programs" earlier in this chapter, relocatable addresses are not available in pure code. Thus, when a caller calls an external procedure (a procedure outside of the compilation unit), the conventions must provide:

- A method for the caller to get the address of the callee
- A method for the callee to locate its own impure data section — that is, to load the DB (A5) register with the address of its DATA$ section

To provide the caller with the address of the callee, the caller's DATA$ section contains the addresses of all external references. The loader inserts the proper absolute addresses when it initializes the impure section at load time. To call an external procedure, the caller loads the entry address from the DATA$ section into a register, then jumps to the subroutine (**JSR**) indirectly through the register.

To allow the callee to locate its own impure data section, the DOMAIN conventions put the entry point to externally-callable routines in the DATA$ section, rather than in the PROCEDURE$ section. A short **prologue** loads the address of the start of the section into a register, then jumps to the pure code. The caller can find the pure code from within the DATA$ section because the loader is free to write the appropriate absolute addresses in the DATA$ section. Figure 1-3 illustrates the process.

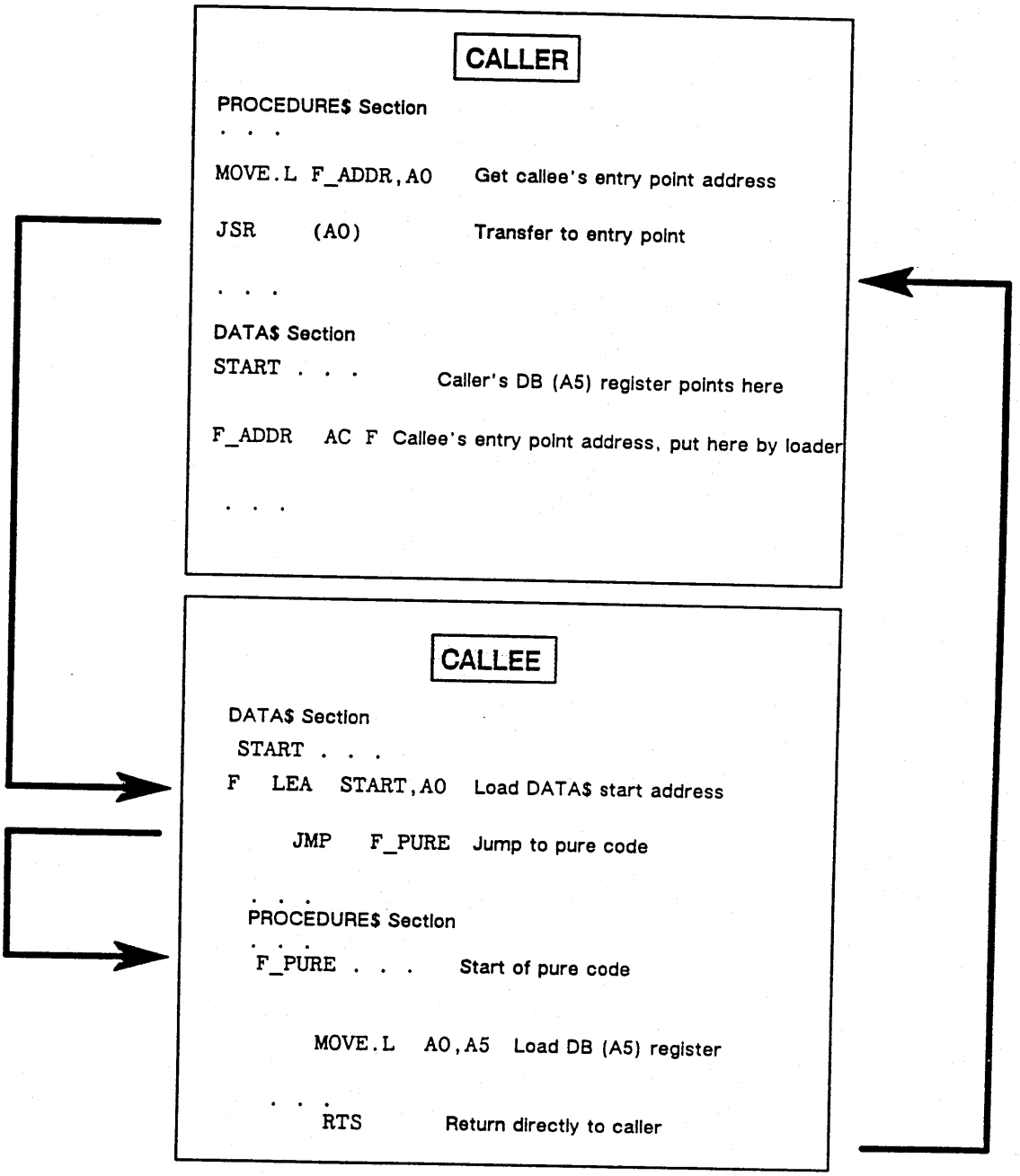


Figure 1-3. External Call Mechanism

Chapter 2

Using ASM

This chapter explains how the DOMAIN assembler (ASM) works and illustrates how to invoke it.

2.1 How ASM Operates

ASM is a two-pass assembler. In first pass, the assembler constructs skeleton code, builds a symbol table, and assigns values to all the labels in your program. During the second pass, the assembler determines the addressing modes in the instructions. At the end of the second pass, the assembler generates the object module file with the **.bin** extension and listing file with the .lst extension. Figure 2-1 illustrates the files that ASM produces from your source file.

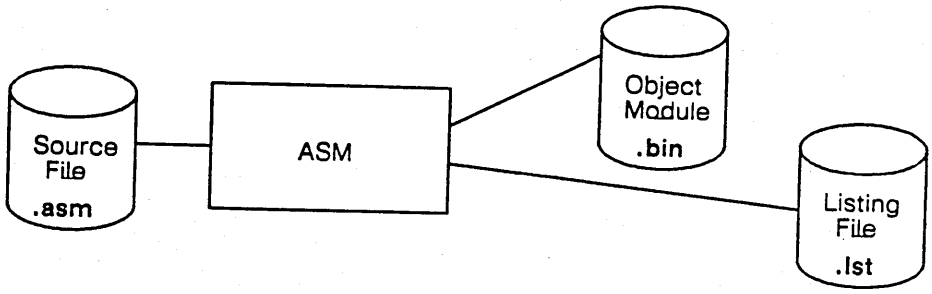


Figure 2-1. Files ASM Produces

Within the program, ASM uses the **location counter** to determine instruction placement. The location counter is a value used for addressing a series of locations. When the assembler processes an instruction, it defines the label with the current value of the location counter. The assembler places the instruction at the current value of the location counter and advances the location counter by the length of the instruction. For example, if label X, defined in a 32-bit instruction on line 1 of your source program, is at location 28476, and label Y is defined on the next line, the location of Y is 28480 (incremented by four bytes).

The location counter consists of a section and an offset within the section. Therefore, the location counter only increments the offset, or number of bytes, in an instruction within the section. When ASM encounters a new section, it resets the location counter to the current offset for that section. Refer to Chapter 4 for more information.

The following pseudo-ops change the offset or location counter:

ORG Changes the offset within a section.

SECT Changes the location counter to any section defined within the module.

PROC Changes the location counter to the section of the predefined procedure frame (usually PROCEDURE$).

DATA Changes the location counter to the section of the predefined data frame (usual DATA$).

You can reference the current value of the location counter in an expression by using the asterisk (*). In the following example, we set *VAR1* to the current value of the location counter.

```
VAR1    EQU     *
```

ASM automatically aligns instructions on even-byte boundaries. Therefore, the location counter is always on an even-byte boundary after ASM processes an instruction. This is because the 68000, which is a byte addressable machine, requires instructions to be aligned on word, or even-byte (.W) boundaries. Also, multi-byte data, such as 16-bit integers and 32-bit integers must be aligned on word boundaries.

NOTE: On the MC68020 and 160/460/660 series nodes, aligning 32-bit integers and floating-point numbers on long word (.L) boundaries increases performance.

The pseudo-ops **DA**, **DC.B**, and **DS.B** can leave the location counter on an odd-byte boundary. If the location counter is at an odd-byte boundary, ASM advances the location counter by 1 before defining the label and processing the instruction. Also, ASM automatically aligns some of the pseudo-ops on even-byte boundaries. Refer to Chapter 4 for more information about pseudo-ops.

2.2 Invoking ASM

Once you have written your assembly program, you can assemble the source file by invoking ASM. To invoke ASM, type the appropriate information on the command line using the following format:

```
$ ASM pathname [options]
```

2.2.1 Pathname

The **pathname** contains the name and location of your source file. The name of the file must end with the suffix **.asm**. However, when you assemble the source file, you do not have to specify the **.asm** extension in the pathname. As explained in the first section of this chapter, when ASM assembles your code, it generates the object module in an object file with the suffix **.bin**. Also, ASM generates the listing file with the suffix **.lst**. Unless you use the **-B** or **-L** options, ASM generates both the binary and listing file in the current working directory. Refer to Figure 2-1 for a graphic representation of the files ASM produces.

For example, if **test1.asm** is the source file name, then you can enter the following information on the command line:

```
$ ASM //beth/asm_programs/test1.asm
```

This causes the assembler to read source statements from the file `//beth/asm_programs/test1.asm` and generate the following binary and listing files in the current working directory:

```
test1.bin          test1.lst
```

2.3.2 Options

ASM contains a number of command line options that enable the assembler to generate or suppress specific listings. Table 2–1 lists the options and provides an explanation of each.

Table 2–1. ASM Command Line Options

Option	Meaning	Default
-L [<pathname>]	Generates an assembly listing (.lst).	✓
-NL	Suppresses an assembly listing.	
-B [<pathname>]	Generates object module (.bin).	✓
-NB	Suppresses binary file.	
-XREF	Generates cross-reference listing	
-NXREF	Suppresses cross-reference listing.	✓
-IDIRpathname [...]	Directs ASM to search a hierarchy of directories for include filenames. The hierarchy applies only to insert pathnames that do not begin with '.', '-', or, '/', '\', or, ''. You can use up to 63 -IDIR options. The compiler first tries to open the specified include filename; if it fails, it prepends -IDIR pathnames to the include filename in the same order as entered on the command line.	
-CONFIG name [...]	Configures sections of code for conditional assembly. Refer to Chapter 4 for more information.	

Chapter 3

DOMAIN Assembly Language

An assembler program consists of a sequence of lines that adhere to a particular format. These lines can contain a comment, a mnemonic instruction, an assembler pseudo-op, or an assembler directive.

This chapter describes the elements of DOMAIN assembly language, including instructions and addressing modes. DOMAIN assembly language typically uses Motorola MC68000 series instruction set notation with little or no variations. However, the differences between Motorola notation and DOMAIN assembly language are detailed in this chapter. You can find more information on the actual 68000 instruction sets in the appropriate MC68000 series manual.

We also provide a brief definition and an example of each of the addressing modes. For more detailed information, refer to the appropriate MC68000 programmer's reference manual.

The topics discussed in this chapter are:

- Source file format
- Instruction format
- Language elements
- Instructions
- Addressing modes

3.1 Source File Format

Every assembler source file begins with either the pseudo-op **PROGRAM** (if you are coding the main program), or the pseudo-op **MODULE** (if you are not coding the main program). Additionally, every file terminates with the pseudo-op **END**.

A DOMAIN assembly language program generally contains two sections: procedure and data. DOMAIN assembly language predefines these sections. Therefore, you do not have to use the **DFSECT** pseudo-op (defined in Chapter 4) to define the sections. The pseudo-ops **PROC** and **DATA** position the location counter to the section.

As we stated in Chapter 1, the procedure section contains position independent code and constant data. All compilers produce pure code in this section. If you do not enter a name for the predefined procedure section in the PROGRAM of MODULE pseudo-op, DOMAIN assembly language predefines the section with the default name **PROCEDURE$**.

The data section contains static read/write data and linkages (address constants to external procedures and data). The data section contains all relocatable references. DOMAIN assembly language predefines this section with the default name **DATA$** if you do not enter a name for it in the PROGRAM or MODULE pseudo-op. Figure 3-1 illustrates the format of a DOMAIN assembly language source program.

```
  PROGRAM name, start-addr [, proc-section name [, data-section name]]

                                or

  MODULE name [, proc-section name [, data-section name]]

  PROC

*pure code and data.  No variables, no relocation.

  DATA

*read/write code and data.  Includes all address constants.

          •
          •
          •
          •
          •


  END
```

Figure 3-1. DOMAIN Assembly Language Source Program Format

3.2 Instruction Format

Most assembler source lines have the following format:

[label] **operator** **operands** **[comments]**

If you use a **label**, it must start in column one. Labels can be up to 32 characters in length. The first character of a label must be either a letter or the underscore character. The **operator** cannot start in column one. Fields are separated by one or more blanks. Most programmers prefer to align the fields in fixed columns, but this is not required.

Anything following the first blank in the **operand** field is interpreted as a **comment.** Therefore, blanks cannot appear within the operand field (except inside quoted strings). Some operators require no operands, in which case everything following the operator is a comment.

An entire line can be used for a comment if you begin the line with an asterisk in column one.

3.3 Language Elements

This section describes the the following elements of DOMAIN assembly language:

- Character set
- Names and values
- Numbers
- Strings
- Register lists
- Expressions

3.3.1 Character Set

The **character set** that the assembler uses consists of upper and lower case letters, digits, and the special characters listed in Table 3-1. Labels, operators, and operands (except quoted strings) are case-insensitive.

Table 3-1. Special Characters

Special Characters	Meaning
*	Indicates a comment line in column one. Also, indicates the current contents of location counter in the operand field.
@	Indicates repetition, as in DC.B 4@0, where 4 bytes are preinitialized to zero.
$	Precedes a hex constant, or a symbol character if not first.

Continued on next page

Table 3–1 (Continued)

Special Characters	Meaning
#	Precedes immediate operands.
%	Preprocessor directive.
, (comma)	Separates arguments within a field.
' (single quote mark)	Delimits strings in a variable field.
. (period)	Size suffix (.B, .W, .L).
: (colon)	Follows labels (optional).
/	Separates register lists.
( )	Sub-expressions, addressing modes.
[ ]	Memory indirect addressing (68020 only).
{ }	Bit field addressing (68020 only).

3.3.2 Names and Values

Names (labels) consist of 1 to 32 characters, each a letter, digit, underscore, or dollar sign. The last character must be a letter or underscore.

Reserved Names

DOMAIN assembly language contains a few reserved symbolic names that you cannot use when defining your own labels. Table 3–2 lists the DOMAIN assembly language reserved names. In the *Processor Type* column, the MC68000 series numbers in parentheses indicate that the names are reserved only under those processors and coprocessors.

Table 3-2. DOMAIN Assembly Language Reserved Names

Reserved Name	Meaning	Processor Type
A0 - A7	Address registers	All
D0 - D7	Data registers	All
SR	Status Register	All
CCR	Conditional Code Register	All
DB	Data frame Base register (A5)	All
SB	Stack frame Base register (A6)	All
SP	Stack Pointer (A7)	All
USP	User Stack Pointer	All
MSP	Master Stack Pointer	All
ISP	Interrupt Stack Pointer	All
DFC	Destination Function Code	(68010, '020)
CACR	CAche Control Register	(68020)
CAAR	CAche Address Register	(68020)
VBR	Vector Base Register	(68010, '020)
FP0 - FP7	Floating-Point registers	(TERN, 68881)
FPIADDR	Floating-Point Instruction ADDRess register	(TERN, 68881)
FPCONTROL	Floating-Point CONTROL register	(TERN, 68881)
FPSTATUS	Floating-Point STATUS register	(TERN, 68881)

Names denote **values**. A value can be a simple number, but it can also be more complex. The following types of values are recognized. Refer to Chapter 4 for descriptions of the pseudo-ops used in the examples.

Absolute — a simple number

```
limit    equ      1440     *The value of 'limit' is the absolute number 1440
```

Section Relative — consists of a section and offset. The value of an instruction or data label is a section-relative value.

```
count    ds.l     1        *The value of 'count' is the value of the location
*                          counter where it is defined.
```

External — the location of an external symbol.

```
          extern.p spline    *The value of 'spline' is the location of the
*                             external symbol of that name.
```

Addressing — names can be equated to addressing expressions, which are described in the section "Addressing Modes" later in this chapter.

```
argl      equ      8(a6)     *The value of 'argl' is 8 bytes past the location
*                             pointed to by address register A6.
```

3.3.3 Numbers

Numbers can be represented in either decimal or hex format. Decimal numbers begin with a digit (0–9). Hex numbers begin with a $ and use digits (0–9) or letters (A through F, uppercase or lowercase) for the remaining characters.

Decimal numbers must be within a range that fits into a 4–byte range, as shown below:

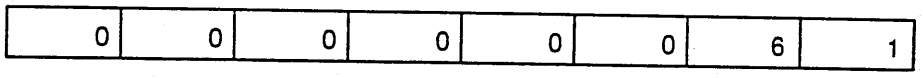
0 to 4294967295 or -2147483648 to 2147483647

The assembler does not generate an error for decimal values that exceed the 4–byte range.

DOMAIN assembly language allows hex numbers to be up to 8 bytes. Unlike the 4–byte decimal limit for decimal numbers, hex numbers allow for immediate double precision floating–point constants.

3.3.4 Strings

Strings use single quotation marks (') to delimit literal information. Limit strings to four characters, unless you use strings in the **DA** pseudo–op variable field. Refer to Chapter 4 for complete information on the **DA** pseudo–op. If the string contains fewer than four characters, the assembler right–justifies the string and fills in the remaining spaces with zeros. For example, the literal string 'a' is equivalent to $61 in hex, as shown.



3.3.5 Register Lists

Register lists enable you to operate on multiple registers. The register list appears in the source or destination fields of the MOVEM instruction. DOMAIN assembly language uses the standard 68000 syntax for register lists, in which Rn is a single register and Rn–Rm is a range of registers (where n is less than m). Use the slash (/) to list more than one register list. For example,

D0–D7/A0–A7

Refer to the appropriate 68000 manual for more information.

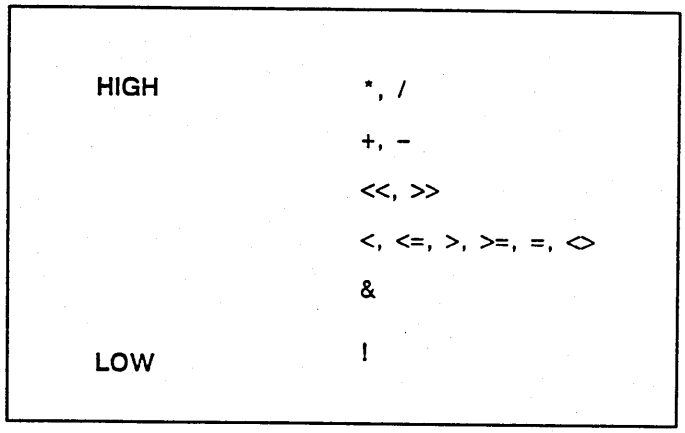
3.3.6 Expressions

Names, numbers, strings, and addressing expressions (described above) can be combined into compound expressions using assembly-time operators. DOMAIN assembly language recognizes four types of operators:

- Arithmetic
- Conditional
- Shift
- Logical

These sections introduce the operators. Before we begin, we look at the rules of operator precedence.

The rules of operator precedence in algebra apply to any expression you write in DOMAIN assembly language. The processor prioritizes operations from high to low and evaluates operations equally from left to right. The following chart illustrates operator precedence from high to low. Parentheses override operator precedence and can be used to control the order of expression evaluation.



Arithmetic Operators

Arithmetic operators enable you to perform mathematical functions. To use arithmetic operators, follow these rules:

- All operands must be 32-bit integers. Therefore, 16-bit integers are extended to 32-bits before the operation.
- All operands, except those used in addition and subtraction, must have an absolute expression type.
- When both operands are absolute expression types, the result is an absolute expression type.

Table 3-3 lists the arithmetic operands, illustrates the operand format, and describes each operand. Refer to the section "Addressing Modes" later in this chapter for more information on the role expression types play in addressing modes.

Table 3-3. Arthimetic Operands

Operand	Description
+ (Opr1 + Opr2)	**Adds** two operands. One of the operands can be absolute, section-relative, external, or an addressing expression. The other operand must be absolute. The result of the two different operand types is the same type as the non-absolute operand.
- (Opr1 - Opr2)	**Subtracts** Opr1 from Opr2. One of the operands can be absolute, section-relative, external, or an addressing expression. The result of the two different operand types is the same type as the non-absolute operand. Also, both operands can be section-relative or external. If so, both operands must refer to the same section or an external. The result type of section-relative or external operands is absolute.
* (Opr1 * Opr2)	**Multiplies** two absolute expression type operands.
/ (Opr1 / Opr2)	**Divides** two absolute expression type operands.

Conditional Operators

Conditional operators test for specific conditions in operands. Table 3-4 lists and describes the conditional operators. Note that the result of all conditional operators is 1 if the condition is true. If the condition is false, the result is 0.

Table 3-4. Conditional Operators

Operand	Description
< (Opr1<Opr2)	Determines if Opr1 is *less than* Opr2.
<= (Opr1<=Opr2)	Determines if Opr1 is *less than or equal to* Opr 2.
> (Opr1>Opr2)	Determines if Opr1 is *greater than* Opr2.
>= (Opr1>=Opr2)	Determines if Opr1 is *greater than or equal to* Opr2.
= (Opr1=Opr2)	Determines if Opr1 is *equal to* Opr2.
<> (Opr1<>Opr2)	Determines if Opr1 is *not equal to* Opr2.

Shift Operators

Shift operators shift data a specified number of bits either to the right or to the left. Table 3-5 lists and describes the shift operators.

Table 3-5. Shift Operators

Operator	Description
<< (Opr1<<Opr2)	**Shift left** Opr1 by the number in Opr2. For example, A<<2 means shift A left 2.
>> (Opr1>>Opr2)	**Shift right** Opr1 by the number in Opr2. For example, B>>3 means shift B right 3.

Logical Operators

Logical operators, or bitwise operators, compare and manipulate bits within the operands. Table 3-6 lists and describes the operators.

Table 3-6. Logical Operators

Operator	Description
& (Opr1&Opr2)	The logical function AND (&) sets the destination bit to 1 if the source bits (in both operands) are set to 1. Otherwise, AND sets the destination bit to 0.
! (Opr1!Opr2)	The logical function OR (!) sets the destination bit to 1 if either or both of the source bits (in the operands) is 1. If both source bits are 0 the destination bit is 0.

3.4 Instructions

A line of source code consists of one or more fields depending on the type of statement you use. A field is defined here as a subdivision of a source line that requires specific information. A source line can contain up to 256 characters. The three kinds of assembly source line statements are:

- Op-codes
- Pseudo-ops
- Directives

An example of an instruction is as follows:

```
TRANSP    MOVE     D0,D1    *Move contents of register D0 to D1.
```

Let's discuss each field in the above format example.

Labels are optional. If you use a label, follow the rules outlined at the beginning of this chapter. If you do not use a label, start the op-code in column two or beyond. Separate the op-code from the operands (in source/destination) with at least one space.

Depending on the type of **op-code**, an instruction can have both a source and destination operand, only a source operand, or *neither* a source *nor* a destination operand. If the instruction has neither, a space following the op-code terminates the op-code field and the assembler regards the rest of the line as a comment. Refer to the "Comments" section later in this chapter for more information.

If the instruction requires both a source and destination operand, separate the operands with a comma (,). Do not allow spaces between the operands. Remember that a space following an operand indicates that the rest of the line is a comment. For example,

```
PAKIT     MOVE     CONT,D0     *Correct format.

STREM     MOVE     CONT, D0    *Incorrect format -- Space before D0 makes it a
                               *comment.
```

3.4.1 Instruction Op-codes

DOMAIN assembly language uses the op-codes defined in the appropriate Motorola processor manuals. Also, for full details on the instruction set, refer to the appropriate Motorola manual.

Variants

Some 68000 instructions use Q(uick), A(ddress), and I(mmediate) variants, such as **ADDQ**, **CMPA**, and **ORI**. With these instructions, DOMAIN assembly language allows you to use the root of the instruction, such as **ADD**, **CMP**, and **OR** to obtain the same results. When you use the root of these instructions, the assembler automatically generates the proper variant if the instruction operands fit the requirements of the variant. For example,

```
MOVE.L    #5,D1
```

is assembled into a **MOVE.Q(UICK)** instruction.

Extensions

Many 68000 instructions, such as **ADD**, operate on byte (**.B**), word (.W), and long (**.L**) operands. The extensions, **.B**, **.W**, and **.L** select the operand length of the instruction. Some examples are: ADD.B, SUB.L, and OR.W. If you do not specify an extension, the default is .W. Appendix B provides a list of 68000 series instructions and legal extensions.

Branch Length Determination

Branch instructions, such as **BRA** and **BGT**, can have **.S** or **.L** extensions. For example, BRA.S is a legal instruction. If the extension is **.S**, the assembler generates a one word branch instruction. If the extension is **.L**, the assembler generates a two word branch instruction. If the branch instruction has no extension, the assembler generates a two word instruction for destinations that are forward references. However, if the destination is not a forward reference, the assembler generates a one word branch instruction only if the destination is within a short branch range (-128 bytes to 127 bytes of the branch instruction).

3.4.2 Pseudo-ops

A **pseudo-op** is a DOMAIN assembly language defined operator that can generate code or data, or can control certain aspects of assembly. Pseudo-ops have a format similar to the instruction format described above. The format for a pseudo-op is:

[label] <pseudo-op> [<variable field>] <comments>

To use pseudo-ops, follow these rules:

- If you use a label, start the label in column one.
- Use one or more spaces between each field.
- If a pseudo-op has no variable field, DOMAIN assembly language considers everything beyond the pseudo-op field to be a comment.
- If a pseudo-op has a variable field, separate each item in the variable field with a comma.

Refer to Chapter 4 for more information on pseudo-ops.

3.4.3 Directives

Assembler **directives** provide information to the assembler about include files and conditional assembly. Directives do not generate machine instruction in assembled code. The format of the directive is:

%<directive> <directive dependent operands>

A percent sign (%) in column one precedes all directives. Skip one or more spaces between the directive and the operands. For more information on directives, refer to Chapter 4.

3.5 Addressing Modes

The addressing modes used in instructions are sometimes specified explicitly by the programmer and sometimes selected by the assembler. For example,

```
move.l    (a2),d1
```

explicitly specifies address register indirect addressing, but

```
move.l    flag,d1
```

only refers to a label and leaves the selection of the appropriate address mode to the assembler. To effectively use the assembler, you need to know the syntax for explicitly specifying addressing modes and to understand the rules which the assembler uses to select implicit addressing modes.

3.5.1 Syntax

DOMAIN assembly language does not always use the addressing mode syntax defined in the Motorola processor manuals. In fact, some addressing modes (PC-relative, for example) cannot be explicitly specified. Table 3-7 lists the DOMAIN assembly language syntax for each of the 68000-family addressing modes. No syntax is listed for those modes that cannot be explicitly specified; the example for those modes illustrate typical instructions for which the mode might be generated by the assembler.

Table 3-7. Addressing Modes Summary

Addressing Mode	Syntax	Example
Data register direct	Dn	move.l D1,D2
Address register direct	An	move.l A1,A2
Address register indirect	(An)	move.l (A1),D1
with postincrement	(An)+	move.b (A3)+,(A0)+
with predecrement	-(An)	clr.w -(A3)
with displacement [1]	d_{16}(An) or (d_{16},An)	move.l 6(A0),DB move.l (6,A0),DB
with index (8-bit disp.) [2]	d_8(An,Xn.SIZE*SCALE) or (d_8,An,Xn.SIZE*SCALE)	tst.w 12(A2,D1.L*4) tst.w (A2,D3)
with index (base disp.) [3]	bd(An,Xn.SIZE*SCALE) or (bd,An,Xn.SIZE*SCALE)	clr.l 500(A2,D3.L*4) clr.l (A2,D3.W) clr.l 500(A2)
Memory indirect [3] post-indexed	([bd,An],Xn.SIZE*SCALE,od)	tst.w ([20,A2],D1.L*2,4) tst.w ([20,A2],4) tst.w ([A2],D1,4) tst.w ([20,A2]) tst.w ([A2])
pre-indexed	([bd,An,Xn.SIZE*SCALE],od)	clr.l ([20,A2,D1.L],4) clr.l ([A2,D1],4)
PC-indirect with displacement with index (8-bit disp.) with index (base disp.) [3]	Not explicitly specified. See Section 3.5.2	beq.s loop move.s edcidc(D1.W),D0
PC memory indirect [3]	Not supported.	
Absolute short Absolute long	Not explicitly specified. See Section 3.5.2	jmp.l test$proc
Immediate	#<data>	move.l #3,D1

Syntax Symbols for Table 3-7

Dn	data register
An	address register
Xn	index register (data or address)
SIZE	index register size, ".W" or ".L"

SCALE	index register scale, 1, 2, 4, or 8
d_8	8-bit (disp)lacement
d_{16}	16-bit (disp)lacement
bd	base displacement (up to 32-bits)
od	outer displacement (up to 32-bits)

Notes for Table 3-7

[1] On a 68020, this syntax can be used with a 32-bit displacement. This will assemble into an address register indirect (base displacement) mode with a suppressed index.

[2] Index scale factors permitted on 68020 only. The default is 1. The default index size is word (.W).

[3] Mode valid on 68020 only.

3.5.2 Address Mode Determination

The assembler determines the proper address mode during its two passes through your program. In the first pass, ASM determines the length of all instructions. This enables ASM to define the values of all the labels in your program. During the first pass, if ASM encounters an instruction operand with an expression containing a forward reference, ASM assumes that the addressing mode for that particular operand requires one word of address extension. During the second pass, if ASM determines that the operand in question requires an address extension other than one word, it displays the error message:

```
ILLEGAL FORWARD REFERENCE
```

The following table illustrates how ASM determines each addressing mode. To use the table, locate the proper expression type in column one. Next, decide whether the expression type meets the conditions described in column two. If the conditions are true, ASM uses the addressing modes listed in column three. For example, if the expression type is absolute and a current USING pseudo-op has an absolute expression, then ASM uses either address register direct or address register indirect with displacement.

Table 3-8. Addressing Mode Determination

If the expression type is . . .	And if...	Then ASM uses this mode . . .
68000 addressing mode	It uses specific address mode	Specified mode.
Section relative	The section is the same as the section of the instruction	PC-relative (unless mode is not allowed for instruction operand).
	A current USING references the same section as the operand	Address register indirect or Indirect with displacement.
	Otherwise	Long absolute.
External	The current USING is the the same as the external	Address register indirect or Indirect with displacement.
	Otherwise	Long absolute.
Absolute	The current USING is an absolute expression	Address register indirect or Indirect with displacement.
	The value of the absolute expression is > -32769 < 32678	Short absolute.
	Otherwise	Long absolute.

3.5.3 Additional Notes

1. If you use the JMP.L instruction, ASM uses long absolute addressing mode. However, the expression type of the operand must be absolute, section-relative, or external.
2. Labels can be indexed, for example:

```
move.w    table(D1.w),D2
```

3. Indexed expressions must resolve either to address register indirect or program counter indirect with index mode. Therefore, the reference must be within the same section (in order for ASM to use the program counter with index), or a current USING must reference the section.
4. ASM does not allow instructions that require relocation in read only sections. Instructions need relocation in the following situations:
   - A section-relative operand referenced with absolute mode.
   - An external operand referenced with absolute addressing mode.
   - Immediates with section-relative or external expressions such as *#FINDER* where FINDER is an external reference.
5. The normal defaults for ASM address mode determination are shown in Table 3-9.

Table 3-9. Normal Case Defaults for Addressing Mode Determination

If the symbol is **Defined in**	And is **Reference from**	Then the **Models. . .**
PROC	PROC	PC-relative
DATA	PROC	A5-relative
PROC	DATA	absolute
DATA	DATA	PC-relative or A5-relative (depends on conditions described in Table 3-8).

Chapter 4

Pseudo-Ops and Directives

This chapter provides a description of each DOMAIN assembly language pseudo-op. The chapter concludes with descriptions of two types of assembler directives: the include file directive, and the conditional assembly directives.

4.1 Pseudo-Ops

In this section, we describe the DOMAIN assembly language pseudo-ops. Pseudo-ops are DOMAIN assembly language implementations of instructions that provide the assembler with special information. Pseudo-ops can be used in conjunction with any MC68000 series instruction set.

AC -- Define address constant

FORMAT

[<label>] AC[.L] <expression>

FIELDS

label — An optional name that refers to the address constant.

AC[.L] — The pseudo-op for Address Constant. The .L extension is the default. No other extension is allowed.

expression — The address constant containing the value of the expression. For example:

```
vfmt_ac   AC        vfmt_$write2
```

where `vfmt_$write2` is a system service routine.

DESCRIPTION

The assembler creates an address constant (AC) at the current value of the location counter. An address constant is a 4-byte quantity that contains the value of the expression. The expression must be absolute, section-relative, or external. If the expression is section-relative or external, the address constant requires relocation. Therefore, the location counter must be set to a read/write section. The AC advances the location counter by 4.

If necessary, the assembler automatically advances the location counter to an even-byte boundary before defining the label and processing the pseudo-op.

Normally, you put AC pseudo-ops in the data frame. Code in the procedure frame loads the address constant into an address register and uses the address register to reference an external symbol or data in another section. The following example illustrates this:

```
            PROC
            PEA       ctrl_str
            MOVE.L    printf_ac,A0
            . . .
            DATA
print_ac    AC        printf
```

CPU -- Define specific hardware requirements

FORMAT

CPU **[68000|68010|68020],[TERN|68881],[68851]**

FIELDS

CPU	The CPU pseudo-op. This pseudo-op tells the assembler which instruction set to use to interpret the floating-point instructions that follow.
68000|68010|68020	The hardware variable field option. You must choose the appropriate option to assemble instructions that are specific to a processor. The default is 68010.
TERN|68881	The FPP (Floating-Point Processor) variable field option. You must choose the appropriate floating-point hardware to properly interpret the in-line floating-point code that follows. The default is TERN. If you are using TERN, you do not have to use this pseudo-op. Because TERN is a processor, it ignores a selection of 68000, '010, or '020.
68851	The PMMU variable field option. You must specify this option to assemble instructions for the PMMU board.

DESCRIPTION

The CPU pseudo-op can be used anywhere in the source file *after* the MODULE or PROGRAM pseudo-op. The CPU pseudo-op defaults are

```
CPU       68010,TERN
```

Thus, if you do not specify the appropriate processors and coprocessors for the source code that follows the CPU pseudo-op, the assembler will either incorrectly interpret the instructions or will not assemble the code. For example, if you write a block of code that uses the 68881 coprocessor instruction set but neglect to use the CPU pseudo-op before that block, the assembler incorrectly interprets the instructions using the default TERN instruction set.

NOTE: In some cases, TERN floating-point instructions use the same assembler op-codes as the 68881, but they assemble into different binary codes.

In a DOMAIN system, it is generally desirable to minimize hardware dependencies so that programs can run on any workstation in the network. This is usually accomplished by restricting the instruction set to the lowest common denominator, typically a 68010 with no floating-point hardware. Note that the floating-point library described in Chapter 7 provides a hardware independent interface for floating-point computations.)

This portability can exact a performance penalty. For maximum performance, you may want to generate multiple versions of the program tailored to different hardware configurations. The CPU pseudo-op is frequently used in conjunction with conditional assembly so that the different versions can share the same source code. Here is a typical example:

```
%var mc68020
. . .
%IF mc68020 %THEN                       if assembling for a 68020
        CPU       68020
        bfextu    (a3){15:2},d1         use bit field instruction, otherwise...
%ELSE
        CPU       68010
        move.l    (a3),d1               shift and mark
        move.l    #15,d0
        lsr.l     d0,d1
        and.l     #$3,d1
%ENDIF
```

If you specify `-config mc68020` on the command line, the MC68020 version assembles; otherwise, the MC68010 assembles. See the discussion of conditional assembly later in this chapter.

NOTE: Unlike the -CPU option of DOMAIN compilers, the CPU pseudo-op *does not* cause the object module to be marked as requiring specific hardware. Thus, the loader will not detect an attempt to run on inappropriate hardware.

DA -- Define ASCII constant

FORMAT

|<label>| DA[.B|.W|.L] '<string>'{,'<string>'}

FIELDS

label — Optional name of character string.

DA — The Define ASCII pseudo-op. The extension .W is the default. The extension of the DA pseudo-op determines alignment. If the extension is **.W** or **.L** and the location counter is on an odd boundary, the location counter advances to an even-byte boundary before defining the label and processing the pseudo-op. If the extension is **.B**, the ASCII string may start on an odd-byte boundary. The extension on the pseudo-op only affects the first string alignment in the variable field. Note that after processing the DA pseudo-op, the location counter can be on an odd-byte boundary regardless of the suffix.

'<string>' — Character strings are enclosed within single quote delimiters. A character string must terminate before the end of the line. If this is not possible, use multiple DA pseudo-op lines. A quote character is represented by two adjacent quote characters. For example, to specify `'ABCDEF'` use `'''ABCDEF'''`. Also, the variable field can contain more than one string if you use commas between the strings. For example, to print `ABCDEF`, use `'ABC','DEF'` in the string variable. Do not skip a space after the comma; the assembler interprets any space in the variable field as the end of the data.

DESCRIPTION

The DA (Define ASCII) pseudo-op sets memory that begins at the current value of the location counter to the ASCII value of the character string. For example:

```
            PEA     p_string
            PEA     c_string
            . . .
c_string    DA.B    '%a%.'
p_string    DA.B    'ABCDEFGH'
```

DATA -- Place subsequent code in the DATA section

FORMAT

DATA

FIELDS

DATA	Sets the location counter to the start of the predefined data section the *first time* the DATA pseudo-op appears. Any subsequent DATA pseudo-op sets the location counter to the current end of the predefined data section. After a DATA pseudo-op, the location counter is always on an even-byte boundary. The DATA pseudo-op is equivalent to a SECT pseudo-op with the section name of the predefined data section (normally DATA$).

DESCRIPTION

The DATA pseudo-op places subsequent code in the data section, which is a read/write, concatenate section.

DC -- Define constant

FORMAT

|<label>| DC{.B|.W|.L} [<count>@]<value>{,...}

FIELDS

label Optional name used to reference the constant.

DC The Define Constant pseudo-op. The extension .W is the default and causes each value to be put into successive words of memory. If the extension is .B, each value is put into successive bytes of memory. If the extension is .L, each value is put into successive long words of memory. The location counter advances to an even-byte boundary, if necessary, before defining a label and processing a DC pseudo-op with a .W or .L extension. If the extension is .B, the location counter may be left on an odd-byte boundary after processing the DC pseudo-op.

count@value A repeat count and value. The repeat count indicates the number of locations to set to the corresponding value in the variable field. The *at* sign (@) separates a repeat count from a value. The expression type of the repeat count and value must be absolute. Use commas to separate multiple items in the variable field. Do not skip a space after the comma because the assembler interprets a space in the variable field as the start of a comment.

DESCRIPTION

The DC (Define Constant) pseudo-op sets memory that begins at the current location counter to the values in the variable field.

The following example defines p_len as 8 and references p_len when the address of the constant (8) is pushed onto the stack.

```
            PEA       p_len
            . . .
p_len       DC.W      8
```

The following example defines a 256-byte table initialized to zero:

```
char_table  DC.B      256@0
```

DEFDS -- Define symbols in descending order

FORMAT

DEFDS **<expression>**

FIELDS

DEFDS — The DEFDS pseudo-op block defines values for names used to reference fields in a modeled data structure. The DEFDS pseudo-op can be used to model a data structure such as a Pascal record or the runtime stack. The DEFDS block does not reserve storage for the data structure and does not alter the value of the location counter.

expression — The expression in the variable field can be any type except a register or immediate value. The expression cannot contain any forward references. The DEFDS expression affects the definition of labels on DS pseudo-ops within the DEFDS block. Also, the DEFDS expression can affect the label on an EQU pseudo-op if the expression in the variable field of the EQU pseudo-op references the location counter.

DESCRIPTION

The DEFDS pseudo-op block spans a number of source lines. The block starts with a DEFDS pseudo-op and extends to an ENDS pseudo-op. Only EQU, DS, ENTRY pseudo-ops and comments can occur between a DEFDS pseudo-op and its matching ENDS pseudo-op. No instructions or other pseudo-ops are permitted.

When the assembler encounters the DEFDS pseudo-op, it saves the current value of the location counter and sets the location counter to the value of the expression in the variable field. Within the block, DOMAIN assembly language processes EQU pseudo-ops normally by defining the label with the value in the expression field. Labels attached to the DS pseudo-op are defined with the value of the location counter minus the number of bytes specified by the DS pseudo-op. After processing the DS pseudo-op, the location counter is *decremented* by the specified number of bytes. When the assembler encounters the corresponding ENDS pseudo-op, it restores the location counter with the value saved at the DEFDS pseudo-op. For instance, the symbol CORE, in the following example, is defined with the value of -4(SB) and Y with the value of -6(SB).

```
        DEFDS   0(SB)
CORE    DS.L    1
Y       DS.W    1
        ENDS
```

DEFS -- Define symbols in ascending order

FORMAT

DEFS **<expression>**

FIELDS

DEFS The DEFS pseudo-op block, like the DEFDS pseudo-op, defines values for names used to reference fields in a modeled data structure.

expression The expression in the variable field can be any type except a register or immediate value. The expression cannot contain any forward references.

DESCRIPTION

The DEFS pseudo-op block spans a number of source lines. The block starts with a DEFS pseudo-op and extends to an ENDS pseudo-op. Only EQU, DS, and ENTRY pseudo-ops and comments can occur between a DEFS pseudo-op and its matching ENDS pseudo-op. No instructions or other pseudo-ops are permitted.

The difference between the DEFS pseudo-op and the DEFDS pseudo-op is the method the assembler uses to compute the value of DS pseudo-op labels within a DEFS block. The assembler sets the DS label to the value of the location counter. Then, the location counter *increments* by the number of bytes specified in the DS pseudo-op.

The following dummy block models a record used as an element of a linked list:

```
        DEFS    (a1)            a1 points to record
DATA    DS.W    1
PRED    DS.L    1
SUCC    DS.L    1
        ENDS
```

DATA is equivalent to 0(a2), PRED to 2(a1), SUCC to 6(a1).

DFSECT -- Define section

FORMAT

<name>	**DFSECT**	**[(overlay,][readonly,][instruction,][zero)]**

FIELDS

<name> The label, or name, of the section. Note that in the DFSECT pseudo-op, the section name appears in the label field while section names appear in the variable field of SECT, MODULE, and PROGRAM pseudo-ops. The DFSECT pseudo-op must appear *before* the first SECT pseudo-op that references that section. The section name in the DFSECT pseudo-op cannot be the same as the names used in the predefined procedure frame or the predefined data frame. However, the section name can have the same name as a label or external.

DFSECT The DFSECT pseudo-op defines a section.

attributes The variable field of the DFSECT pseudo-op can contain any of the following section attributes: **overlay, readonly, instruction,** and **zero.** Use commas to separate attributes. If you do not specify the overlay attribute, the section is a concatenate section. If you do not specify the read-only attribute, the section is read/write. For more information on attributes, refer to Appendix F, *The Object Module.*

DESCRIPTION

The DFSECT pseudo-op defines a section. Two sections are predefined in a DOMAIN assembly language module and thus do not need a DFSECT pseudo-op: PROC and DATA. You can define additional sections to reference FORTRAN *COMMON* blocks from assembly language or to control working set. Use one DFSECT pseudo-op per section.

The DFSECT pseudo-op has no effect on the location counter. The SECT pseudo-op positions the location counter on a section.

DROP -- Discontinue use of base address register

FORMAT

DROP **An**

FIELDS

DROP The DROP pseudo-op ends the effect of the corresponding USING pseudo-op in the address mode determination of instruction operands.

An The variable field contains the name of an address register. The DROP pseudo-op matches the previous USING pseudo-op with the same address register.

DESCRIPTION

The DROP pseudo-op ends the effect of the corresponding USING pseudo-op in the address mode determination of instruction operands. DOMAIN assembly language sets up an implicit USING of A5 to start the predefined data frame. You can counteract this by using a `DROP DB` (reserved name for A5).

```
            program     sample,commence
*The following example program runs entirely in the data section.  The DROP
*pseudo-op is used to force the assembler to use PC-relative addressing
*instead of  the default DB-relative addressing.
            DATA
            DROP DB
operand1    ds.l    1
operand2    ds.l    1
sum         ds.l    1
commence    move.l  #5,operand1
            move.l  #3,operand2
            move.l  operand1,d0
            move.l  operand2,d1
            add.l   d1,d0
            move.l  d0,sum
            rts
            END
```

DS -- Define uninitialized storage

FORMAT

|<label>| DS[.B|.W|.L] <expression>

FIELDS

label	Optional name used to reference the defined storage.
DS	The DS pseudo-op reserves storage. The extension on the DS pseudo-op and the expression in the variable field determine the amount of storage reserved. If the extension is .W or .L, the location counter advances to an even-byte boundary (if necessary) before defining the label and processing the pseudo-op. The amount of reserved storage depends on the extension and the expression. If the extension is .B, the amount of reserved storage is the value of the expression. If the extension is .W, the amount of storage is twice the value of the expression. If the extension is .L, the amount of storage is four times the value of the expression.
expression	The expression type must be absolute. Also, the expression cannot contain any forward references.

DESCRIPTION

The DS pseudo-op reserves storage. Unlike the DC pseudo-op, which reserves and initializes storage, the DS pseudo-op has no effect on the contents of storage. DOMAIN assembly language advances the location counter over the reserved storage. A DS pseudo-op with a .B extension may leave the location counter on an odd-byte boundary.

The following example defines 32 bits of storage for CAPRICORN and moves the absolute value of 4 into that space.

```
CAPRICORN       DS.L      1
                . . .
                MOVE.L    #4,CAPRICORN
```

EJECT -- Begin new page in listing

FORMAT

EJECT

FIELDS

EJECT	The EJECT pseudo-op. Causes the assembler to eject the current forms and begin printing on a new page. The EJECT pseudo-op can precede the PROGRAM or MODULE pseudo-op.

DESCRIPTION

After the printer completes printing a full page of code, it automatically ejects to the top of the next page. To control printing output, use the EJECT pseudo-op in the operation field where you want the printer to eject the current page and begin printing on the next page. In the following example, EJECT causes the procedure push_em to start on a new page in the listing file:

```
          . . .
          RETURN    pop_em
          EJECT
push_em   PROCEDURE #12,D2-D3
          . . .
```

END -- Signal the end of an assembly language module

FORMAT

END

FIELDS

END	The END pseudo-op marks the end of an assembly language module.

DESCRIPTION

The END pseudo-op must be the last statement in a program. DOMAIN assembly language does not allow multiple modules per source file. Thus, DOMAIN assembly language ignores anything written after the END pseudo-op, as shown in the following example.

```
start           jmp     code_start
                ac      start
                dc.w    0
vfmt_ac         ac      vfmt_$write2
                END
continue_rtn    Any code following END is ignored.
```

ENDS -- End a DEFDS or DEFS

FORMAT

ENDS

FIELDS

ENDS	The ENDS pseudo-op marks the end of a DEFDS or a DEFS block.

DESCRIPTION

The ENDS pseudo-op must match a previous DEFDS or DEFS pseudo-op, as shown below.

```
        defs    (A0)
core    ds  1
y       ds  2
        ends
```

ENTRY -- Declare a global symbol definition

FORMAT

ENTRY[.D|.P|.F] **<name>**

FIELDS

ENTRY — The ENTRY pseudo-op makes a name defined in the assembly language module accessible to other assembly language modules or high-level language modules. The **.D** extension indicates data, **.P** indicates procedure, and, **.F** indicates function. The extension affects the use code field of the global definition in the object module.

name — The name in the variable field of the ENTRY pseudo-op must be defined in the assembly language module with either an absolute or section-relative value. The assembler makes the name a global definition in the object module.

DESCRIPTION

The ENTRY pseudo-op defines data, procedures, or functions that outside modules can use. To make a name an entry point, include the name in the variable field of the ENTRY pseudo-op. For example,

```
          ENTRY.F   SUB1      *Entry point for function called SUB1.
SUB1      EQU       *
```

NOTE: The use code (.D, .P, or .F) does not currently affect binding or execution.

EQU -- Define a symbol to equate with the value of the expression

FORMAT

<label> EQU <expression>

FIELDS

<label> Name of the symbol to contain the value of the expression.

EQU The EQU pseudo-op assigns to the label the value of the expression that follows.

expression The value you want to assign to the label. The expression can be any type; however, DOMAIN assembly language does not allow the expression to be a forward reference. To reference the location counter, use an asterisk in this field. The location counter is not affected by the EQU pseudo-op.

DESCRIPTION

The EQU pseudo-op equates the label with the value contained in the expression field. For example,

```
AAA     EQU     0(SB)
BBB     EQU     AAA+20
```

equates AAA to 0(A6). Then, EQU equates BBB to the contents of AAA+20, or 20(SB). Additionally, consider the following example, which equates code_start with the current location of the location counter.

```
code_start      EQU     *
```

EXTERN -- Declare a name as a global defined outside the module

FORMAT

EXTERN[.D|.P|.F] <name>

FIELDS

EXTERN	The EXTERN pseudo-op allows the assembly language module to reference global names defined in other modules written in assembly language or in a high-level language. The **.D** extension indicates data, **.P** indicates procedure, and, **.F** indicates function. The extension affects the use code field of the global definition in the object module.
name	The name in the variable field declares the global defined outside the module. This name must not be defined within the module.

DESCRIPTION

The EXTERN pseudo-op references data, a procedure, or a function outside the module. The EXTERN pseudo-op generates a global reference in the object module. If the DOMAIN assembly language module defines data, a procedure, or a function, the module contains an ENTRY pseudo-op along with a name. However, if the DOMAIN assembly language module uses outside data, procedures, or functions, the module contains an EXTERN pseudo-op along with a name. For example, PROG1 contains:

```
ENTRY.F   MATH_CALCULATIONS
```

PROG2, which calls `Math_Calculations`, contains:

```
EXTERN.F  MATH_CALCULATIONS
```

A program that references an external symbol usually defines an address constant for it. For example,

```
A$CALC    DA        MATH_CALCULATIONS
```

LIST -- Enable listing

FORMAT

LIST

FIELDS

LIST	The LIST pseudo-op allows the current line and subsequent lines below the pseudo-op to print in the listing file.

DESCRIPTION

The LIST pseudo-op allows the current line and subsequent lines below the pseudo-op to print in the listing file. This pseudo-op counteracts the effect of the NOLIST pseudo-op. Use the LIST and the NOLIST pseudo-ops together to inhibit the listing of insert files. The following example illustrates how to use the LIST pseudo-op.

```
          MODULE INVERT
          NOLIST
%INCLUDE  'FPP.INS.ASM'
          LIST
          . . .
```

You can nest NOLIST and LIST pseudo-ops.

MODULE -- Begin a module

FORMAT

MODULE **<name>[,<procname>,<dataname>]**

FIELDS

MODULE	The MODULE pseudo-op is the heading for an assembly language module. Every DOMAIN assembly language compilation unit must begin with either a **MODULE** or **PROGRAM** heading.
name	The assembler puts this name into the name field of the global information header in the object module. The name has no relationship to other names in the module; therefore, you can use this name as a section name or a label name elsewhere in the program.
procname	Section name for the predefined procedure frame. The name must not be the same as the name in a DFSECT pseudo-op. If you do not use this option, the assembler names the section PROCEDURE$.
dataname	Section name for the predefined data frame. The name must not be the same as the name in a DFSECT pseudo-op. If you do not use this option, DATA$ is the default name for the section.

DESCRIPTION

The MODULE pseudo-op is the heading for DOMAIN assembly language modules which contain only subroutines. Every DOMAIN assembly language program must contain either one MODULE or PROGRAM heading. Except for comments, EQU, and EJECT, no code can appear before the MODULE or PROGRAM pseudo-op in the source file.

NOLIST -- Suppress a listing

FORMAT

NOLIST

FIELDS

NOLIST	The NOLIST pseudo-op prevents the lines following the pseudo-op from printing in the listing file.

DESCRIPTION

The NOLIST pseudo-op inhibits the lines following the pseudo-op from printing in the listing file. This pseudo-op counteracts the effect of the LIST pseudo-op. Use the LIST and the NOLIST pseudo-ops together to inhibit the listing of insert files. The following example illustrates how to use the NOLIST pseudo-op.

```
          MODULE INVERT
          NOLIST
%INCLUDE  'FPP.INS.ASM'
          LIST
          . . .
```

When you invoke the assembler, the code under the LIST pseudo-op prints in the listing (**.lst**) file. However, the code beneath the NOLIST pseudo-op does not print.

ORG -- Set location counter

FORMAT

ORG **<expression>**

FIELDS

ORG The ORG pseudo-op reoriginates the value of the location counter during assembly.

expression The expression type in the variable field must be absolute or section-relative. If the expression is absolute, the assembler sets the offset field of the location value to the offset field value. If the expression is section-relative, the section of the location counter must be the same as the section of the expression. Forward references are not allowed.

DESCRIPTION

The ORG pseudo-op changes the offset part of the location counter to the value of the expression. The ORG pseudo-op does not change the section of the location counter but changes the position within the section. Other pseudo-ops, such as DATA, PROC, and SECT change the value of the location counter to a different section.

PROC -- Place subsequent code in the PROCEDURE section

FORMAT

PROC

FIELDS

PROC	The PROC pseudo-op sets up the procedure section of the assembly program.

DESCRIPTION

The PROC pseudo-op sets the location counter to the current end of the predefined procedure section. The assembler automatically sets the location counter to the start of the predefined procedure section at the start of an assembly language module. After the PROC pseudo-op, the location counter is always on an even-byte boundary. The PROC pseudo-op is equivalent to a SECT pseudo-op with the section name of the predefined procedure frame. The following is an example of the PROC pseudo-op.

```
MODULE . . .
.
.
.
DATA            switch to DATA section
.
.
.
PROC            switch back to PROC section
.
.
.
```

PROCEDURE – Generate standard procedure entry sequence

FORMAT

```
<label>  PROCEDURE  ['<debug-name>'|"<debug-name>"]
                                    [,#-<local-size>]
                                    [,<saved-regs>]
                                    [,NOCODE]
                                    [,STANDARD|NONSTANDARD]
                                    [,NOSTACK]
                                    [,NOXEP]
                                    [,ALTENTRY]
```

NOTE: The arguments can be listed in any order.

FIELDS

label	A label is required. In addition to labeling the generated code, the label links the PROCEDURE pseudo-op to corresponding RETURN pseudo-ops and provides the default procedure name in debugging information.
debug_name	The name to be placed in the debug information generated for this procedure. This is the name that appears in tracebacks. The name can be an arbitrary string of up to 32 characters; the single and double quoted forms are identical. If omitted, <label> is used.
#-<local-size>	Bytes of local variable storage to allocate in the stack frame. This becomes the argument to the generated LINK instruction. The value must be an even, negative number. If omitted, zero is assumed.
saved-regs	List of registers to be saved, in the same format that is used in a MOVEM instruction. This does not include floating-point registers. If omitted, no registers are saved.
NOCODE	Suppresses code generation. This allows the pseudo-op to be used to specify debugging information only.

NOTE: The remaining arguments affect debugging information only.

NONSTANDARD	Specifies that this procedure does not adhere to standard stack frame conventions. More specifically, it usually means that the procedure does not LINK. This flag is set by default if you specify NOCODE.
STANDARD	Specifies that this procedure adheres to standard stack conventions. This is the default if you let PROCEDURE generate the entry sequence. However, if you specify NOCODE but manually code a standard prologue, you should use this option.
NOSTACK	Specifies that the procedure does not use the stack at all. This is a special case of nonstandard stack usage. In effect, this indicates that the return address will remain at the top of the stack during the procedure's execution. Debugging and traceback tools can exploit this knowledge.
NOXEP	Specifies that the procedure does not have an external entry prologue. An external entry prologue is the DATA section code that sets the DB register before jumping to the pure code.

ALTENTRY Specifies that this procedure has alternate entry points.

DESCRIPTION

The PROCEDURE pseudo-op generates a standard procedure entry sequence and specifies debugging information to be associated with it.

The PROCEDURE pseudo-op generates the following code sequence:

```
<label> link    sb,#-<local_size>
        movem.l <saved-regs>,-(sp)
```

A RETURN pseudo-op referencing the label generates a standard procedure exit sequence corresponding to the entry:

```
movem.l -n(sb),<saved_regs>
unlink  sb
rts
```

The SB offset n in the MOVEM instruction is calculated as

```
n = <local_size> + 4*(number of saved registers)
```

Refer to the entry for the RETURN pseudo-op for more information.

The assembler generates a minimal set of debugging information so that DEBUG, TB, and related tools can determine which procedure a program is executing. Information about variable names, line numbers, etc., is not available to debuggers.

In assembly language, a *procedure* is not always well-defined. All code between a PROCEDURE pseudo-op and the next PROCEDURE pseudo-op (or the end of the file) is treated as a part of the *procedure* in the debugging information. If no PROCEDURE pseudo-ops are present in a module, the assembler constructs a dummy *procedure* for each pure section and names it *module-name:section-name*. If you use any PROCEDURE pseudo-ops, you should ensure that all pure code is covered by a procedure definition.

Procedure entry and exit sequences are discussed in Chapter 6, which contains examples of the use of PROCEDURE and RETURN.

PROGRAM -- Begin a program

FORMAT

PROGRAM **<name>,<start>[,<procname>,<dataname>]**

FIELDS

PROGRAM The PROGRAM pseudo-op is the heading for an assembly language main program module. Every DOMAIN assembly language file must begin with either a **PROGRAM** or **MODULE** heading. PROGRAM identifies the file as containing a main program.

name The assembler puts the program name into the name field of the global information header in the object module. The name has no relationship to other names in the module; therefore, you can use this name as a section name or a label name elsewhere in the program.

start The start name is a section-relative expression. It indicates where to begin the execution of the program. This name is separated from the program name by a comma. The value is put into the object module start address.

procname Section name for the predefined procedure frame. The name must not be the same as the name in a DFSECT pseudo-op. If you do not use this option, PROCEDURE$ is the default name for the section.

dataname Section name for the predefined data frame. The name must not be the same as the name in a DFSECT pseudo-op. If you do not use this option, DATA$ is the default name for the section.

DESCRIPTION

The PROGRAM pseudo-op is the heading for the main program in DOMAIN assembly language. Every DOMAIN assembly language program must contain either a MODULE or PROGRAM heading. Except for comments, EQU, and EJECT, no code can appear before the MODULE or PROGRAM pseudo-op in the source file. Below is an example of a DOMAIN assembly language program pseudo-op.

```
            PROGRAM ABC,START_HERE
            PROC
code_start  equ *
            link    sb,#0
                    . . .
            DATA
START_HERE  lea     start_here,a0
            jmp.l   code_start
```

Note that the program name is ABC. Also note that the start name is START_HERE. When the program runs, it begins at START_HERE.

RETURN – Generate standard procedure exit sequence

FORMAT

[<label>] PROCEDURE <procedure-label>

FIELDS

label	Optional label for the exit code.
procedure-label	The label of a PROCEDURE pseudo-op in the same source file.

DESCRIPTION

The RETURN pseudo-op generates a standard procedure exit sequence corresponding to the entry sequence generated by a PROCEDURE pseudo-op. See the description of PROCEDURE for further details.

SECT -- Set location counter to end of named section

FORMAT

SECT **<name>**

FIELDS

SECT The SECT pseudo-op changes the value of the location counter to a different section.

name This is the section name. The section name must also appear in a DFSECT pseudo-op preceding the SECT pseudo-op or must be the name of a predefined procedure or data frame.

DESCRIPTION

When the assembler encounters a SECT pseudo-op, it changes the location counter to the current end of the named section. Subsequent code and data are assembled into the new section.

The PROC and DATA pseudo-ops are equivalent to SECT pseudo-ops referencing the predefined procedure and data sections.

Subsequent lines of code within each section increment the value of the offset. The following example illustrates the SECT pseudo-op.

```
          MODULE COMMON
          ENTRY
abc       dfsect   overlay
          SECT     abc
a         ds.l     2
b         ds.l     1
c         ds.l     1
          data
common    move.l   #-5,c
          rts
          end
```

USING -- Specify base register address of expression

FORMAT

USING **An,<expression>**

FIELDS

USING The USING pseudo-op tells the assembler which base register to use. It affects the address mode determination of instruction operands. USING affects all instructions that follow it until the assembler encounters a DROP pseudo-op with the same address register. Then, the DROP pseudo-op counteracts the USING address mode determination.

An The name of an address register.

expression The expression must be external, section-relative, or absolute. The USING pseudo-op tells the assembler to assume the address register contains the expression when determining the address mode of instructions. Thus, if the expression is the name of an external, and the instruction following USING references that external, DOMAIN assembly language assumes that the address register in USING contains the address of the external. Then, the assembler selects an address mode using the specified address register.

DESCRIPTION

The USING pseudo-op generates no instructions and simply tells the assembler which base register to use. You must load the base register using the appropriate instructions to ensure that the address register described in the USING pseudo-op contains the value of the expression when the instructions within the range of the USING pseudo-op are executed. The assembler sets up an implicit USING of A5 (DB) to the start of the predefined data frame. You can counteract this with a DROP DB. A DROP pseudo-op must appear before another USING pseudo-op that specifies the same address register.

4.2 Directives

DOMAIN assembly language directives in the source program enable the assembler to perform certain tasks without generating code. We introduce the directives in this section and provide some examples. The topics are:

- Include files
- Conditional assembly

4.2.1 Include Files

The %INCLUDE directive allows the assembler to insert source text from another file into the assembling module. This enables you to access other files without having to bind them into your source program. Thus, you can easily access common files for many of your DOMAIN assembly language programs. The %INCLUDE directive format is:

```
%INCLUDE 'pathname'
```

The pathname is the name of the file from which the assembler inserts source statements. You must enclose the pathname in single quotation marks (') as shown in the format above. The assembler inserts the contents of the file at the point where the %INCLUDE directive appears. We demonstrate how an include file operates in the following example. The main program uses the %INCLUDE directive to retrieve data from a data file called /programs/asm_programs/stringdata.

```
            PROGRAM CALLSTRING,START
            EXTERN vfmt_$write2
            PROC
set_up      equ *
            link     sb,#0
            move.l   db,-(sp)
            move.l   a0,db
            pea      p_len
            pea      p_string
            pea      c_string
            move.l   vfmt_ac,a1
            jsr      (a1)
            addi.w    #12,sp
            move.l   (sp)+,db
            unlk     sb
            rts
%include '/programs/asm_programs/stringdata'
            DATA
start       lea      start,A0
            jmp.l    set_up
vfmt_ac     ac       vfmt_$write2
            END
```

When the main program reaches the include directive, it accesses the data from the `stringdata` data file, shown below.

(STRINGDATA data file)

```
c_string    da.b    '%a%.'
p_string    da.b    'DATA OK!'
p_len       dc.w    8
```

NOTE: Each time you modify text in the data file, you must reassemble the main program before executing it again.

You can nest insert files up to a 16 level limit; that is, your program can access an insert file that contains other %INCLUDE file directives. The **-IDIR** command line option establishes search rules for insert files. The option enables you to select alternate pathnames at assembly time. Refer to "Invoking ASM" in Chapter 1 for detailed information.

4.2.2 Conditional Assembly

Conditional assembly allows you to mark sections of code for conditional processing, and later select the marked sections at assembly time with the **-CONFIG** command line option. The **-CONFIG** option invokes conditional processing. In **conditional processing**, you begin by declaring valid attributes (in your source program) with the **%VAR** directive. Then, depending on the evaluation of the attribute name or conditional expression (called the **predicate**), the assembler determines whether to assemble or ignore marked sections of code.

Invoking Conditional Assembly

DOMAIN assembly language supports conditional processing in the same manner as the DOMAIN compilers. To invoke conditional processing in DOMAIN assembly language, use the following format:

```
$ asm sourcefile_name -CONFIG name1 name2 . . .
```

An attribute is set to TRUE if it appears as an argument to the -CONFIG option, or is enabled by the **%ENABLE** directive. Otherwise, the attribute is FALSE. Therefore, depending on whether a specific predicate evaluates TRUE or FALSE, the assembler determines whether the text following an **%IF** *predicate* **%THEN** condition or the text following an **%ELSEIF** *predicate* **%THEN** condition assembles.

For example, we have expanded the program shown in the *Include File* section to allow conditional processing based on whether the predicate `string` is TRUE or FALSE. First, the %VAR directive declares `string` as a valid attribute. Then, selection of either the `stringdata` data file or the `nostringdata` data file occurs, depending on whether we declare `string` in the -CONFIG option list at assembly time.

```
%VAR string
%IF string %then
%include '/programs/asm_programs/stringdata'
%ELSE
%include '/programs/asm_programs/nostringdata'
%ENDIF
```

{NOSTRINGDATA data file}

```
c_string    da.b    '%a%.'
p_string    da.b    'NO DATA!'
p_len       dc.w    8
```

If we invoke conditional processing, as shown below,

```
$ asm stringcall.asm -CONFIG string
```

`string` is TRUE. The program includes the `stringdata` data file and prints `DATA OK!` However, if we do not invoke conditional processing (do not use the **-CONFIG** option at assembly time), `string` is FALSE. In this case, the program includes `nostringdata` and prints `NO DATA!`

Forms of Predicates

Several of the directives take a **predicate**. A predicate can consist of special variables (declared with the %VAR directive) and the optional Boolean keywords NOT, AND, or OR.

Before introducing the conditional assembly directives, we look at the forms a predicate can take to alter results. Table 4–1 illustrates the forms.

Table 4–1. Predicate Forms

Predicate Form	Meaning
name	Attribute name
NOT predicate	TRUE if predicate is FALSE
predicate AND predicate	TRUE if both predicates are TRUE
predicate OR predicate	TRUE if either or both predicates TRUE
(predicate)	Grouping

Conditional Assembly Directives

Conditional assembly uses many directives. Table 4–2 lists all the conditional assembly directives and their meanings. Following the table, we describe each directive in detail.

Table 4–2. Assembler Directives

Directive	Meaning
%IF *predicate* %THEN	Assembles code up to the next %ELSE, %ELSEIF, or %ENDIF directive, if and only if the predicate is true.
%ELSE	Follows an %IF...%THEN condition. Assembles code up to the next %ELSEIF or %ENDIF if the predicate is false.
%ELSEIF *predicate* %THEN	Assembles code up to the next %ELSE, %ELSEIF, or %ENDIF directive, if and only if the predicate is true.
%ENDIF	Indicates the end of a program's conditional compilation area.
%IFDEF *varname* %THEN	Verifies whether a variable (varname) was previously declared with a %VAR directive.
%ELSEIFDEF *varname* %THEN	Verifies whether additional variables (varnames) were declared with a %VAR directive.
%VAR *varname*	Declares attribute names that you can use with predicates in the assembler directive.
%ENABLE *varname*	Sets an assembler attribute name to TRUE.
%CONFIG	Special predicate evaluates to TRUE when you use the -CONFIG command line option.
%ERROR *'string'*	Prints string as error message whenever you assemble the code.
%WARNING *'string'*	Prints string as warning message whenever you assemble the code.
%EXIT	Directs the assembler to stop conditionally processing the file.

We describe each of the directives on the pages that follow.

DESCRIPTION

%CONFIG is not a directive but a predeclared attribute name. You can only use the %CONFIG directive as a predicate. The assembler sets the %CONFIG directive to TRUE if you use the **-CONFIG** option in your assembler command line. If you do not, the assembler sets the %CONFIG directive to FALSE.

EXAMPLE

In this example, if we do not set either the `average` or `mode` attributes to TRUE, a warning that we've written into the code displays: `This program will find the median of the numbers.`

```
%VAR  average mode


          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ELSEIF mode %THEN

CODE that finds the mode of the numbers

%ELSEIF not %CONFIG %THEN
%WARNING('This program will find the median of the numbers.')

CODE that finds the median of the numbers
          .
          .
          .
%ENDIF
```

NOTE: Do not attempt to declare a %CONFIG directive in a %VAR directive.

%ELSE

DESCRIPTION

The %ELSE directive is used in conjunction with the %IF (or %IFDEF) *predicate* %THEN condition. The %ELSE directive and its subsequent block of code up to next %ENDIF assembles only if the predicate evaluates to FALSE.

EXAMPLE

Suppose you want to assemble a block of code that calculates the median of a group of numbers if you do not find the average. In this case, `average`, the predicate, is FALSE. Indicate the following in your program:

```
%VAR  average

          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ELSE

CODE that finds the median of the numbers
          .
          .
          .
%ENDIF
```

To find the median of the numbers, you can omit the **-CONFIG** option at assembly time.

%ELSE

DESCRIPTION

The %ELSE directive is used in conjunction with the %IF (or %IFDEF) *predicate* %THEN condition. The %ELSE directive and its subsequent block of code up to next %ENDIF assembles only if the predicate evaluates to FALSE.

EXAMPLE

Suppose you want to assemble a block of code that calculates the median of a group of numbers if you do not find the average. In this case, `average`, the predicate, is FALSE. Indicate the following in your program:

```
%VAR  average
          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ELSE

CODE that finds the median of the numbers
          .
          .
          .
%ENDIF
```

To find the median of the numbers, you can omit the **-CONFIG** option at assembly time.

DESCRIPTION

The %ELSEIF *predicate* %THEN directive is used in conjunction with the %IF *predicate* %THEN condition. The %ELSEIF *predicate* %THEN condition is the same as ELSE followed by IF . . .THEN except that ENDIF terminates all IF. . . ELSE condition statements.

EXAMPLE

Suppose you want to assemble either a block of code that calculates the mean of a group of numbers or a block of code that calculates the mode of the group of numbers. If both predicates evaluate to FALSE, you still find the median of the numbers. To perform this, make the following statements in your program:

```
%VAR  average mode


          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ELSEIF mode %THEN

CODE that finds the mode of the numbers

%ELSE

CODE that finds the median of the numbers
          .
          .
          .
%ENDIF
```

At assembly time, you can declare either or both the mean and the mode TRUE using the -CONFIG option, or you can declare either or both of the attributes TRUE using the %ENABLE directive. However, if you declare both TRUE, the assembler assembles the code for average because it satisfies the first condition.

DESCRIPTION

The %ELSEIFDEF *varname* %THEN condition verifies whether additional variables were previously declared with the %VAR directive. DOMAIN assembly language issues an error message if you declare a variable more than once. %ELSEIFDEF enables you to avoid the error. Use the %ELSEIFDEF *varname* %THEN condition when you are using insert files that may declare the same variable.

EXAMPLE

The following example uses an insert file that may declare the variables `average` and `mode`. To ensure that you do not define the variables twice, use the %ELSEIFDEF *varname* %THEN condition, as shown:

```
%INCLUDE '/stats_calc/basics'

%IFDEF not(average) %THEN
%VAR  average
%ELSEIFDEF not(mode) %THEN
%VAR mode
%ENDIF
```

%ENABLE *varname1 varname2 ... varnameN*

DESCRIPTION

The %ENABLE directive sets an attribute to TRUE. This directive performs the same function as the **-CONFIG** command line option. You must declare the attribute with the %VAR directive before using the %ENABLE directive. If you do not use the %ENABLE directive or do not assemble the code with the **-CONFIG** option, the attribute is set to FALSE.

EXAMPLE

The following example declares the attributes `average` and `mode` as predicates and sets them to TRUE.

```
%VAR  average mode
%ENABLE average mode
```

NOTE: The assembler issues the following error message if you attempt to set the same variable TRUE more than once:

```
(PreProc) Name "attribute name" is already enabled
```

DESCRIPTION

The %ENDIF directive stops the conditional processing of a particular area of code. Each conditional processing area of code must end with an %ENDIF directive.

EXAMPLE

The following example illustrates the placement of the %ENDIF directive.

```
%VAR  average mode


          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ELSEIF mode %THEN

CODE that finds the mode of the numbers

%ELSE

CODE that finds the median of the numbers
          .
          .
          .
%ENDIF
```

DESCRIPTION

The %ERROR directive causes the assembler to print a string as an error message. Also, it does not generate an executable object. Always place this directive on a line by itself.

EXAMPLE

In our example, if we do not set the average attribute to TRUE, an error that we've written into the code displays: CANNOT EXECUTE THIS PROGRAM.

```
%VAR  average mode


          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ELSE
%ERROR 'CANNOT EXECUTE THIS PROGRAM.'
          .
          .
          .
%ENDIF
```

Thus, when you attempt to assemble this program without the -CONFIG option, the following error message is displayed:

```
(0026) %ERROR 'CANNOT EXECUTE THIS PROGRAM'
**** CONDITIONAL PROCESSOR ERROR
** errors, ASM rev 7.18
```

DESCRIPTION

The %EXIT directive causes the assembler to stop conditionally processing the file.

EXAMPLE

In this example, from the beginning of an insert file, if the variable graphics_2d is already defined, the assembler stops reading the file.

```
%IFDEF graphics_2d %THEN
%EXIT
%ENDIF
%VAR graphics_2d
      .
      .
      .
```

DESCRIPTION

If the predicate is TRUE, the assembler assembles the code up to the next %ELSE, %ELSEIF, or %ENDIF directive.

EXAMPLE

Suppose you want to assemble a block of code that calculates the mean of a group of numbers. After choosing the attribute name `average` as the predicate, you indicate the following in your program:

```
%VAR  average

          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ENDIF
```

To average the numbers, either use the %ENABLE directive within your source program or use the **-CONFIG** option at assembly time.

DESCRIPTION

The %IFDEF *varname* %THEN directive verifies whether a variable was previously declared with the %VAR directive. The assembler issues an error message if you declare a variable more than once. %IFDEF enables you to avoid the error. Use the %IFDEF *varname* %THEN condition when you are using insert files that may declare the same variable.

EXAMPLE

A common use of %IFDEF is to prevent multiple inclusions of the same insert file, which can be a problem if insert files contain %INCLUDE directives themselves. The following directives inserted at the beginning of an insert file cause the assembler to ignore the directive if it has already been included:

```
%IFDEF graphics_2d %THEN
%EXIT
%ENDIF
%VAR graphics_2d
      .
      .
      .
```

NOTE: The %IFDEF condition is TRUE if the variable is declared, regardless of whether it has been enabled through a -CONFIG option or %ENABLE directive. Contrast this to the %IF directive, in which an error occurs if the variable has not been declared.

DESCRIPTION

The %VAR directive allows you to declare variables and attribute names that are used as predicates in your source program. You cannot use an attribute as a predicate unless you declare it with the %VAR directive.

EXAMPLE

The following example declares the attributes `average` and `mode` as predicates.

```
%VAR  average mode
```

NOTE: Do not attempt to declare the same attribute more than once; the assembler will issue an error. To avoid the error, use the %IFDEF or %ELSEIFDEF directive conditions.

DESCRIPTION

The %WARNING directive causes the assembler to print a string as a warning message. Unlike the %ERROR directive, however, assembly continues. Always place this directive on a line by itself.

EXAMPLE

In our example, if we do not set the average attribute to TRUE, a warning message that we've written into the code displays: USE THE -CONFIG OPTION.

```
%VAR  average mode


          .
          .
          .
%IF average %THEN

CODE that averages the numbers
          .
          .
          .
%ELSEIF mode %THEN
%WARNING 'USE THE -CONFIG OPTION.'

CODE that finds the mode of the numbers
          .
          .
          .
%ENDIF
```

Thus, when you attempt to assemble this program without the -CONFIG option, the following warning message is displayed:

```
(0026) %WARNING 'USE THE -CONFIG OPTION'
**** CONDITIONAL PROCESSOR ERROR
** errors, ASM rev 7.18
```

Chapter 5

The Listing File

The listing file (.lst) is one of the two files that ASM automatically generates when you assemble your source program. In Chapter 2, we introduced the other file: the object module file (**.bin** or binary file). Also, we discussed the source file (**.asm**). The **listing file** contains your source code along with the offsets, the hexadecimal code or data assembled, the line numbers, and any error messages that ASM generates during the assembly procedure. The listing file can be helpful when you are debugging your program.

We divide this chapter into three sections:

- Examining the listing file
- Special symbols
- Cross-reference listing

We begin by examining an actual listing of a program. Use the listing for reference throughout the chapter.

5.1 Examining the Listing File

The following example illustrates a sample listing. Unless you use the -NL command line option to suppress the listing, ASM automatically generates a listing when you assemble a program. Suppose you create a program in a file called `abc.asm` and you check the directory in which you assemble your source program `abc.asm`. You see the following entries:

```
abc.asm          abc.bin          abc.lst
```

To display the listing file, read the file with the **.lst** extension. In the following example of the listing file, we have added boldface headers to highlight our discussion of each column within the sample listing:

```
Offset   Object Code      Line #          Source Code

                          (00001)              PROGRAM ABC,START
                          (00002)              EXTERN  vfmt_$write2
                          (00003)              PROC
         00000000         (00004) code_start   equ     *
000000:  4E560000         (00005)              link    sb,#0
000004:  2F0D             (00006)              move.l  db,-(sp)
000006:  2A48             (00007)              movea.l  a0,db
000008:  487A0028         (00008)              pea     p_len
00000C:  487A001B         (00009)              pea     p_string
000010:  487A0014         (00010)              pea     c_string
000014:  206D000A         (00011)              move.l  vfmt_ac,a0
000018:  4E90             (00012)              jsr     (a0)
00001A:  DEFC000C         (00013)              add.w   #12,sp
00001E:  2A6EFFFC         (00014)              movea.l -4(sb),db
000022:  4E5E             (00015)              unlk    sb
000024:  4E75             (00016)              rts
000026:  256125           (00017) c_string     da.b    '%a%'
000029:  4142434445       (00018) p_string     da.b    'ABCDEFGH'
         464748
000032:  0008             (00019) p_len        dc.w    8
                          (00020)              DATA
000000>  41FAFFFE         (00021) start        lea     start,a0
000004>  4EF900000000     (00022)              jmp     code_start
00000A>  00000000         (00023) vfmt_ac      ac      vfmt_$write2
                          (00024)              END
No errors, ASM rev **** X.XX
```

Figure 5-1. Sample ASM Listing File

5.1.1 Offset

The first column in the listing is the **offset** column. The offset contains the location or address of the instruction relative to the start address of the section (for example, PROC or DATA). Note that the offset begins at location 000000 twice within the listing (PROC and DATA). This is because each section begins at location zero. Within a section, you can find the size of an instruction by subtracting the instruction's offset from the offset of the instruction on the next line. For example,

```
000004:  2F0D             (00006)              move.l  db,-(sp)
000006:  2A45             (00007)              movea.l  a0,db
```

We find that `move.l db,-(sp)` is two bytes long. The entire PROC section is 60 bytes long.

Note the colon (:) following some of the offsets. DOMAIN assembly language uses this symbol to identify types of sections corresponding to the location counter. Refer to the "Special Symbols" section within this chapter for more information.

5.1.2 Object Code

The second column contains the **object code**. You can get information such as the op-code value, the length of the operand, the addressing mode, and the data in an instruction. For example,

```
000029: 4142434445     (00018) p_string     da.b     'ABCDEFGH'
        464748
```

The object code represents the string 'ABCDEFGH' in hexadecimal ASCII. Note that the object code only shows what the object module loads in the location. Also, note that each group of object code digits provides information on the length of the instruction.

```
000004: 2F0D           (00006)
000006: 2A48           (00007)
```

NOTE: DOMAIN assembly language does not include symbols in the object code for relocatable code and external symbols.

5.1.3 Line Number

The third column contains the **line number**. When you assemble your program, if an error occurs, DOMAIN assembly language prints out the source line and error message to standard output. For example,

```
 000018: 4E90          (00012)                jsr      (a0)
**** ILLEGAL SYMBOL IN ADDRESS FIELD
```

Also, the listing file contains the error below the source line. In the cross-reference listing, which we describe later in the chapter, DOMAIN assembly language lists the line numbers at which a symbol is defined and used.

5.1.4 Source Code

The fourth column contains your **source code** as you entered it. This enables you to reference your source code as you read the object code.

5.2 Special Symbols

The following table lists the special symbols that DOMAIN assembly language uses in the offset column to identify the section of the location counter.

Table 5-1. Special Symbols In Listing File

Symbol	Meaning
!	Location counter set to non-section-relative expression.
:	Read only section.
>	Read/write section.

5.3 Cross-Reference Listing

The cross-reference listing provides you with an alphabetical listing of all labels and other symbols that you use in your program. When you are debugging your program, the cross-reference listing can provide helpful information about symbols.

To append the cross-reference listing to your listing file, enter the **-XREF** command line option when you assemble your program. For example,

```
$ ASM abc.asm -xref
```

ASM generates a cross-reference listing and appends it to the bottom of the standard listing. Figure 5-2 illustrates a sample listing. We have added boldface headers as reference points to the discussion of each column, which follows.

```
Symbol          Offset      Section          Line Numbers

CODE_START   00000000   PROCEDURE$    0004D 0022
C_STRING     00000026   PROCEDURE$    0010  0017D
P_LEN        00000032   PROCEDURE$    0008  0019D
P_STRING     00000029   PROCEDURE$    0009  0018D
START        00000000   DATA$         0001  0021D
VFMT_$WRITE2  00000000  VFMT_$WRITE2       0002D 0023
VFMT_AC      0000000A   DATA$         0011  0023D
No errors, ASM rev X.XX
```

Figure 5-2. Sample Cross-Reference Listing

5.3.1 Symbol

The first column contains the names of the symbols in your program.

5.3.2 Offset

The second column contains the offset of where the symbol is defined from the beginning of the section.

5.3.3 Section

The third column contains the name of the section or the external in which the symbol is defined.

5.3.4 Line Numbers

The fourth column contains all the line numbers on which the symbols occur. Note that the D following the line number indicates the line number in which each symbol is defined.

PART 2: Run-time Conventions

Chapter 6

Calling Conventions

Assembly language gives you much greater control over the run-time behavior of a program than you can obtain through a compiler. However, your freedom is constrained by the program's need to interact with other software. This chapter describes the calling conventions that you must understand and follow in order to write assembly language code that calls or is called by external procedures.

Although calls are the most direct form of interaction between a procedure and its environment, they are not the only kind. Procedures must not only accept standard calls and leave the machine in a correct state on return, but they must also maintain a consistent stack and register state throughout execution. To conclude that a procedure (that is correct at its interfaces) should be free to do whatever it wants internally, is to ignore several considerations, such as stack unwinding and debugging and analysis tools.

STACK UNWINDING. There are several circumstances where programs can make a nonlocal transfer of control that requires **unwinding** part of **the call stack**. These nonlocal transfers of control include: invocation of a cleanup handler, set with *pfm_$cleanup*, Unix *longjmp*, and Pascal *non-local goto*.

Stack unwinding depends on a known, consistent stack state. An assembler program that uses the stack or registers in a nonstandard way can cause the unwind operation to fail.

DEBUGGING AND ANALYSIS TOOLS. Several programming tools examine the registers and stack of an executing program. These include DEBUG, TB, HPC, and DPAT. A program that does not adhere to standard run-time conventions can cause these tools to produce incorrect or incomplete results.

We provide an overview of the contents of this chapter in Chapter 1. If you have not yet read that material, you should do so before continuing with this chapter. Also, refer to the examples at the end of this chapter for additional information.

NOTE: Calling conventions changed at SR9.5; the pre-SR9.5 conventions are described in Appendix E.

In this chapter, we discuss the following topics:

- Register usage
- Stack frame
- Argument passing conventions
- Calling a procedure
- Procedure prologue and epilogue
- Addressing the data section
- Floating-point registers
- Examples

6.1 Register Usage

As we noted in Chapter 1, three of the 68000 address registers are assigned 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. Instructions such as JSR and PEA support this conventional 68000 stack usage.

A6 is the Stack Base (SB), which points to a fixed position in the stack frame of the currently active routine. Access to local variables and arguments is relative to this register.

A5 is the Data Base (DB), which points to the start of the impure data section (usually DATA$) associated with the active routine. Access to static data is relative to DB. Routines that do not require access to a data section need not set this register.

Registers D0 and A0 are used to return function results. We describe function results in more detail below.

Procedures are required to preserve the contents of registers A2–A6 and D2–D7. A7, the stack pointer, is also normally preserved. However, it is legal for a procedure to return with an SP less than its value on entry, that is, with additional data pushed onto the stack. In particular, **pfm_$cleanup**, builds a cleanup handler record on the stack (it also adds a dummy argument for its caller to pop). Example 4, at the end of this chapter, illustrates how to use a cleanup handler. Also, refer to *Programming With General System Calls* for more information about cleanup handlers.

Registers A0–A1 and D0–D1 are *free-for-all* registers that do not have to be saved. Short procedures can often get by with only these registers and thus avoid saving and restoring registers.

Except for DB, SB, and SP, stack unwinding does not restore the values of the A and D registers. From the caller's view this is equivalent to saying that the procedures pfm_$cleanup and setjmp do not preserve registers. Therefore, callers of these routines must save and restore the registers. Note that such a caller must save and restore all preservable registers, whether or not it uses them itself.

Conventions involving floating-point registers on workstations that have them are described in the section "Floating-Point Registers" below.

6.2 Stack Frame

Figure 6-1 shows the structure of a procedure stack frame in the most general case. Not all of the fields are present in every actual stack frame.

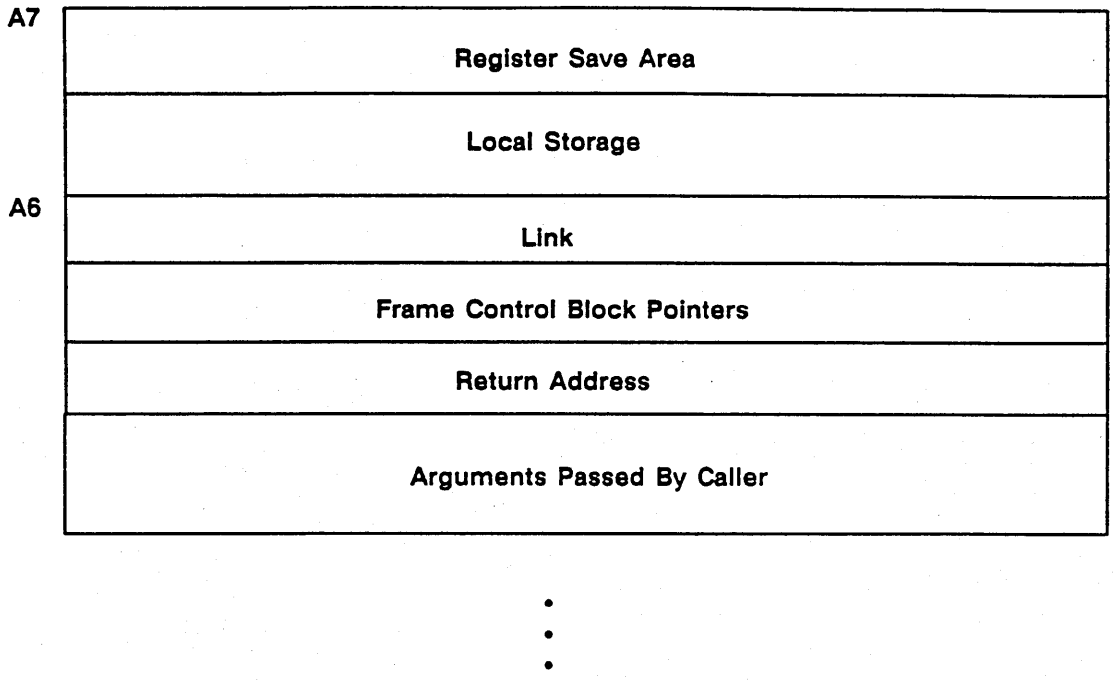


Figure 6–1. Stack Frame Format

These are the fields, from bottom to top (the order in which they are pushed):

ARGUMENTS. The caller pushes arguments onto the stack. They are pushed in reverse order, so the first argument is at the top of the stack. Argument passing conventions are discussed in the next section.

RETURN ADDRESS. The JSR or BSR instruction pushes this address onto the stack and transfers control to the routine.

FRAME CONTROL BLOCK POINTERS. Frame control blocks (FCBs) provide a place to store supplementary control information associated with the frame. If an FCB is present, its address *plus 1* is pushed onto the stack. FCBs must begin at even addresses; therefore, the address plus 1 is odd. This distinguishes the FCB pointer from the return address, which is always even.

The stack frame format allows any number of FCBs and also permits an FCB pointer to be zero. Currently, an FCB is used only to store information about floating–point registers that the procedure saves. Therefore, a procedure that saves and restores floating–point registers has one FCB pointer, and a procedure that does not save and restore floating–point registers has none. We describe the format of the floating–point register FCB itself below in the section "Floating–Point Registers".

LINK. 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.

LOCAL STORAGE. This field provides storage for the procedure's local variables, temporaries, etc.

SAVED REGISTERS. If the procedure changes any of registers A2–A5, D2–D7, or FP2–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 stack frame is popped prior to returning.

6.3 Argument Passing Conventions

Procedures pass arguments by value, in which case a copy of the argument value is pushed onto the stack; or, you can pass procedure arguments by reference, in which case the address of the actual argument is pushed. The mode is determined by the language and the declared attributes of the procedure, as detailed in this section.

6.3.1 Pascal

In Pascal, all arguments of externally called routines are normally passed by reference, regardless of the declared mode (*in, out, in out,* or *var*). Note that this is true even for arguments with call-by-value semantics (default). 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 called procedure is responsible for ensuring 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 4 bytes or less in size and are not arrays, are passed by value. Arguments larger than 4 bytes and all arrays are passed by reference.

If you use the compiler option -ALIGN (true by default), padding is added when arguments smaller than 4 bytes are pushed onto the stack. This ensures that every argument begins at a stack address that is a multiple of 4. For example, passing a 2-byte integer by value generates the following code:

```
SUBQ.L  #2,SP              adjust SP
MOVE.W  arg,-(SP)          push 2-byte arg
```

If you specify -NALIGN, padding is not inserted and arguments may not be aligned on long word boundaries. Arguments are always aligned to even addresses.

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 is free to optimize argument passing as it sees fit. Currently, internal routines are treated the same as *val_param* routines.

6.3.2 FORTRAN

In FORTRAN, all arguments are passed by reference.

6.3.3 C

In C, all arguments are normally passed by value.

The C language specifies that certain conversions are automatically made when passing a value as an argument. Table 6-1 lists the argument type conversions in C.

Table 6-1. Argument Type Conversions in C

Actual Argument Type	Converted To
char, short	int (= long in DOMAIN C)
unsigned char, unsigned short	unsigned int (= unsigned long)
float	double
array of T	pointer to T
function returning T	pointer to function returning T

Note that converting an array to a pointer has the same effect as passing the array by reference, although C does not view it in those terms.

The -ALIGN and -NALIGN compilation options have the same effect in both C and Pascal. Note, however, that arguments shorter than 4 bytes are rarely passed to C functions due to the above rules. They are possible, for small structs.

To permit C programs to communicate with procedures written in other languages, the *std_$call* attribute specifies that arguments are passed by reference rather than by value. The *std_$call* attribute applies only to calls made from C programs to external routines. Routines written in C always expect their arguments to be passed by value. See the *DOMAIN C Language Reference* manual for more details about *std_$call*.

6.3.4 Function Results

Function results are returned in a register if they are 4 bytes or less in size. In C and FORTRAN, 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 4 bytes are returned via a "hidden" argument. The caller of a function that returns a result larger than 4 bytes must push an additional argument on the stack. This additional argument is the address of the location where the result is stored. The argument logically precedes all others (that is, it is the last one pushed onto the stack).

6.3.5 Data Representation

Assembly language routines that communicate with high-level language routines need to know the internal representation of the compiler-generated data. The DOMAIN compiler reference manuals describe internal data representations for each language.

6.3.6 Library Routines

All arguments of system library routines are generally passed by reference. This allows the routine to be called from Pascal, FORTRAN, and (by using *std_$call*) 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 routines intended to be called from multiple languages, beware of data types that do not have direct analogs in all languages, such as Boolean.

6.4 Calling a Procedure

Calling a procedure involves three steps:

1. Push the arguments onto the stack.
2. Transfer control to the procedure, pushing the return address.
3. After the procedure returns, pop the arguments off the stack.

Here is a typical calling sequence for an external routine whose arguments are passed by reference:

```
* pgm_$get_args(argc, argv)
*
                PEA         argv                Push 2nd arg address
                PEA         argc                Push 1st arg address
                MOVE.L      a$pgm_$get_args,A0  Get address of entry point
                JSR         (A0)                Call it
                ADD.L       #8,SP               Pop args
                . . .
                DATA
                . . .
                EXTERN.P    pgm_$get_args       Declare external procedure
a$pgm_$get_args AC          pgm_$get_args       Address of pgm_$get_args
        . . .
```

Note that since this call is to an external routine (in this case, an installed global library routine), it uses an address constant stored in the data section. The MOVE.L instruction, which fetches the address, assembles into a DB-relative address mode. Refer to Example 2, at the end of this chapter, for more a complete example.

Internal calls usually can be made with a PC-relative BSR instruction. Here is an example of an internal call that passes some of its arguments by value:

```
* fill_array(table, size, 10)
*
                PEA         10                  Push value 10
                MOVE.L      size,-(SP)          Push value of size
                PEA         table               Push address of table
                BSR         fill_array          Call it
                ADD.W       #12,SP              Pop args
```

This example illustrates a couple of coding tricks:

- For pushing constant values onto the stack, PEA is often faster and shorter than an equivalent MOVE.
- Because all address register arithmetic is done in long mode, ADD.W is equivalent to ADD.L, but is two bytes shorter. If the constant is less than or equal to 8, the assembler assembles an ADDQ instruction in either case.

6.5 Procedure Prologue and Epilogue

Before a procedure begins its real work, it must do some preparatory housekeeping:

- Build a stack frame
- Save its caller's registers
- Establish the ability to address the data section

We defer discussion of the third point until the next section. For now, assume that the procedure doesn't need to access static data (true for many routines.)

The following is an example of the standard prologue code that accomplishes point one and two: building a stack frame and saving its caller's registers:

```
beth    LINK      #-20,SB              Build stack frame (20 bytes local storage)
        MOVEM.L   A2-A3/D2-D5,-(SP)    Save registers that will be changed.
```

At the end of the procedure, the above steps are reversed:

```
        MOVEM.L   -44(SB),A2-A3/D2-D5  Restore saved registers
        UNLK      SB                   Pop stack frame
        RTS                            Return
```

Note the use of SB-relative addressing to restore the saved registers. The 44-byte offset is the sum of the 20 bytes of local storage plus the 24 bytes needed to save the six registers. You could use (SP)+ addressing if you are certain that SP will not change; however, the SB-relative form is safer.

The PROCEDURE and RETURN pseudo-ops generate these standard code sequences automatically. Using these pseudo-ops the above examples reduce to:

```
beth            PROCEDURE   #-20,A2-A3/D2-D5
                  . . .
                RETURN      beth
```

Some performance-critical assembly language routines omit the LINK and UNLK instructions. If you want to omit them, please note the following:

- Programs like DEBUG, TB, and DPAT are confused by the lack of a standard stack frame. In a traceback, usually the *caller* of a routine with a *missing* stack frame is skipped. Using the PROCEDURE pseudo-op with the NONSTANDARD attribute helps, but may not completely solve the problem.
- A routine that doesn't LINK should not change SB at all. SB must always point to a valid stack frame, even if it's not the current one.
- A routine that does not allocate a stack frame must still observe proper stack discipline, that is, never try to use storage above SP. Space above SP may be overwritten by the operating system during fault handling.

6.6 Addressing the Data Section

We now consider how a procedure loads the DB (A5) register with the address of its impure data section. You may recall from Chapter 1 that the complexity in this arises from the combination of

- Position-independent code, which means that the address of the data section is not known until the program is loaded
- Pure, read-only code, which prevents the loader from writing the data section address into the pure code section where it is needed

Also, Chapter 1 shows how the problem is solved entering procedures that need to set DB through the data section. An **eXternal Entry Prologue** (XEP) in the data section loads the section address into a register and then jumps to the pure code. Here is a typical entry sequence:

```
                DATA
start_data$     EQU         *                   Start of data section for module
                . . .
                DATA
                ENTRY.P     beth                Declare entry point for beth
beth            EQU         *                   XEP for beth starts here
                LEA         start_data,A0       Load data section address
                JMP.L       beth$proc            Jump to pure code
                PROC
beth$proc       EQU         *                   Start of pure code for beth
                LINK        #-8,SB              Standard entry prologue
                MOVEM.L     A5/D2-D3,-(SP)
                MOVE.L      A0,DB               Set DB
                . . .
```

The XEP cannot load DB directly because the caller's DB has not yet been saved. Thus, A0, which need not be preserved, temporarily holds the value.

The LEA instruction in the XEP can use either PC-relative or absolute addressing. The JMP instruction uses an absolute address; the ".L" suffix is required here.

A PROCEDURE pseudo-op can be used to generate the LINK and MOVEM instructions in the prologue. It does not generate the MOVE.L instruction. For example:

```
foo$proc        PROCEDURE   'foo',#-8,A5/D2-D3
                MOVE.L      A0,DB
```

Note that an explicit debug name has been specified, so that tracebacks will show 'foo' rather than 'foo$proc'. Refer to Example 3, at the end of this chapter. Also, for more information about the PROCEDURE pseudo-op, refer to Chapter 4.

Procedures in the same module normally share a single data section. Therefore, internal procedures do not require an XEP. If an externally-callable routine is called internally, the XEP can be short-circuited as in the following example:

```
                MOVE.L      DB,A0               Load A0 with data address
                BSR         foo$proc            Call pure code directly
```

6.7 Floating-Point Registers

On workstations with hardware floating-point units, procedures are required to preserve floating-point registers FP2-FP7. Procedures that change any of these registers save them with an FMOVEM instruction before or after the MOVEM instruction, which saves the A/D registers. (On TERN workstations, FMOVEM is not available and an equivalent sequence of FMOVEs is used instead.)

However, there is an important difference between the treatment of floating-point and A/D registers: floating-point registers are restored by stack unwinding nonlocal transfers of control. Therefore, routines that call *pfm_$cleanup* or *setjmp* need not save all the floating-point registers, which would add considerable overhead as well as introduce machine-dependency problems.

In order to restore the floating-point registers, the stack unwinder needs to know which registers the procedure saved and where it saved them. This information is contained in a **Frame Control Block** (**FCB**). As we discussed in the section "Stack Frame," earlier in this chapter, an FCB pointer (the pointer actually contains the address plus 1) in the stack frame points to this block. The format of the FCB is shown in Figure 6-2.

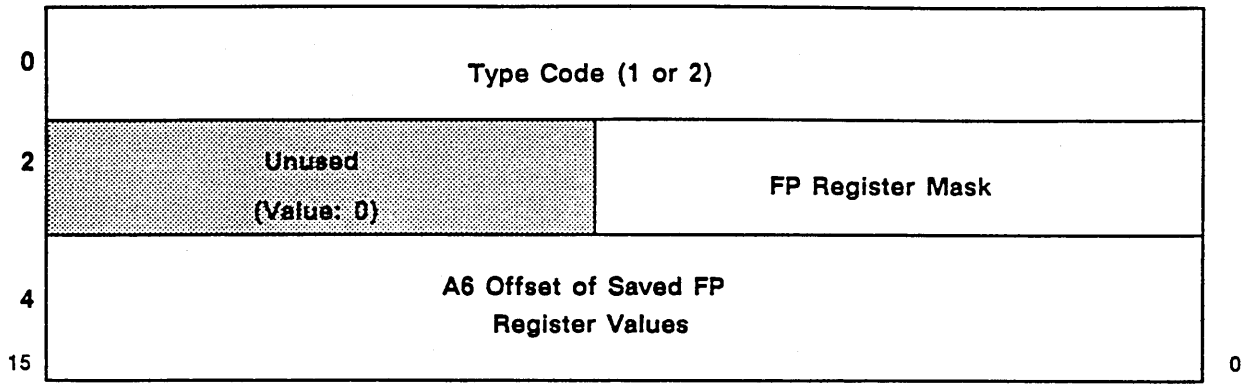


Figure 6-2. MC68881 FP Frame Control Block

The FCB contains the following fields:

TYPE CODE. This 16-bit word identifies the type of floating-point processor for which register information is being saved:

1 - MC68881 floating-point coprocessor
2 - TERN floating-point processor

The remainder of the block format is identical for the two processors.

REGISTER MASK. This field identifies the registers that are saved. Bit 0 (the least significant bit) corresponds to FP7 and bit 7 to FP0. The high order byte of this word is unused and must be zero.

SAVE AREA OFFSET. This is a 32-bit integer that gives the offset of the beginning (low address) of the floating-point register save area from SB.

Here is an example of the entry sequence of a procedure that saves some MC68881 floating-point registers:

```
floating_p      PEA         fp_reg_info+1       Push FCB pointer
                LINK        #-8,SB              Build stack frame
                MOVEM.L     A2/D2-D5,-(SP)      Save A/D registers
                FMOVEM.X    FP2-FP4,(-SP)       Save floating-point registers
                . . .
* Floating-point Register FCB
                . . .
fp_reg_info     DC.W        1                   Type = MC68881
                DC.W        $38                 Mask = 2#00111000 (FP2-FP4)
                DC.L        -64                 Save area offset
                . . .
```

Because the FCB is constant, it can be placed in the procedure section. The save area offset in the FCB is computed as

```
     8 bytes local storage allocated by LINK
+ 20 bytes for storing the 5 registers A2/D2-D5
+ 36 bytes for storing 3 floating-point registers in extended format
```

Example 5 in the next section shows how a dummy section modeling the stack frame can ease the chore of computing stack offsets.

It is necessary to save floating-point registers by pushing them onto the stack, rather than copying into an already allocated part of the stack frame. This allows a stack unwinder to verify that the registers have actually been saved by comparing SP with SB + offset. Consider, for example, a program that receives a quit fault after executing the LINK but before the FMOVEM.

The corresponding exit sequence for this example is:

```
FMOVEM.X    -64(SB),FP2-FP4     Restore floating-point registers
MOVEM.L     -28(SB),A2/D2-D5    Restore A/D registers
UNLK        SB                  Pop stack frame
ADD.L       #4,SP               Pop FCB pointer
RTS                             Return
```

Note that the PROCEDURE and RETURN pseudo-ops do not generate this form of prologue/epilogue code. For more information about these pseudo-ops, refer to Chapter 4.

6.8 Examples

The remainder of this chapter presents some complete examples of assembly language modules.

Note that a useful source of further examples is the expanded listings generated by the compilers. Be aware, however, that the syntax of the expanded listing files is a pseudo-assembly language which is not always legal DOMAIN assembly language input.

EXAMPLE 1 illustrates a simple procedure that does not require register saving or access to the data section.

```
            MODULE      example_1
            entry.p     get_int             Declare entry point
*************************************************************************
*   GET_INT
*
*   Convert ASCII string containing decimal value to binary integer.  Stops
*   at first character which is not a decimal digit.  Does not handle signs
*   or overflow.
*
*   Pascal:     function get_int (in str: univ string): integer32;
*
*   C:          long get_int (str)
*               char *str;
*
*   FORTRAN:    function get_int (str)
*               integer*4 get_int
*               character*(*) str
*
*   Stack frame:
*
*           +---------------+
*      SP-->| local temp    | -4 (offset from SB)
*           +---------------+
*      SB-->| link          |  0
*           +---------------+
*           | return addr   | +4
*           +---------------+
*           | addr(str)     | +8
*           +---------------+
*************************************************************************
             PROC                          *** PROCEDURE SECTION ***

get_int     equ         *
            link        sb,#-4          Link & allocate local storage
            move.l      8(sb),a0        Get arg = address of start of string
            clr.l       d0              Initialize result

Loop        clr.l       d1              Get next character
            move.b      (a0)+,d1
            sub.b       #'0',d1         Convert digit to binary
            blt.s       Done            If result is negative..
            cmp.b       #9,d1           ..or greater than 9..
            bgt.s       Done            ..then character wasn't a digit
            add.l       d0,d0           Multiply result so far by 10...
            move.l      d0,-4(sb)       temp = result*2
            asl.l       #2,d0           result*8
            add.l       -4(sb),d0       result*10
            add.l       d1,d0           Add in new digit
            bra.s       Loop            Back for next character

Done        unlk        sb              Pop frame
            rts                         Exit

            END
```

EXAMPLE 2 is a procedure that saves registers, accesses the data section, and calls an external library routine.

```
            MODULE      example_2
            entry_p     get_int

*******************************************************************************
*
*   GET_INT
*
* Stack frame:
*
*           +---------------+
*      SP-->| saved D2      | -8 (offset from SB)
*           +---------------+
*           | saved DB      | -4
*           +---------------+
*      SB-->| link          |  0
*           +---------------+
*           | return addr   | +4
*           +---------------+
*           | addr(str)     | +8
*           +---------------+
*******************************************************************************
            DATA                                *** DATA SECTION ***
data_start  equ         *                       DB will point here

* External Entry Prologue (XEP)                 Callers enter here
get_int     lea         data_start,a0           Load data section address
            jmp.l       get_int$proc            Jump to pure code

            extern.p    vfmt_$ws                Declare external routine
a$vfmt_$ws  ac          vfmt_$ws                Address constant for external routine
           PROC                                ***** PROCEDURE SECTION *****

get_int$proc equ        *
            link        sb,#0                   Link (no local storage)
            movem.l     d2/db,-(sp)             Save registers we change
            move.l      a0,db                   Set DB pointer
            move.l      8(sb),a0                Get arg = address of start of string
            clr.l       d0                      Initialize result

Loop        clr.l       d1                        Get next character
            move.b      (a0)+,d1
            sub.b       #'0',d1                   Convert digit to binary
            blt.s       Done                      If result is negative..
            cmp.b       #9,d1                     ..or greater than 9..
            bgt.s       Done                      ..then character wasn't a digit
            add.l       d0,d0                     Multiply result so far by 10...
            bvs.s       Overflow                   (check for overflow at each step)
            move.l      d0,d2                      temp = result*2
            asl.l       #2,d0                      result*8
            bvs.s       Overflow
            add.l       d2,d0                      result*10
            bvs.s       Overflow
            add.l       d1,d0                     Add in new digit
            bvc.s       Loop                      Back for next character, unless...
```

Continued

Continued

```
* Result overflowed! Output a message to errout stream and return -1.
* Call vfmt_$ws(stream_$errout, '*** (get_int) overflow%');
Overflow    pea         errmsg              Push addr of error message string
            pea         errout              Push addr of stream id
            move.l      a$vfmt_$ws,a0       Get address of vfmt_$ws
            jsr         (a0)                Call it
            addq.l      #8,sp               Pop args
            move.l      #-1,d0              Return -1 result

Done        movem.l     -8(sb),d2/db        Restore saved registers
            unlk        sb                  Pop frame
            rts                             Exit
* Constant Data

errout      dc.w        3                   stream_$errout
errmsg      da.b        '*** (get_int) overflow%.'
            END
```

EXAMPLE 3 is identical to example 2 except that it uses PROCEDURE and RETURN pseudo-ops.

```
            MODULE      example_3
            entry.p     get_int
*******************************************************************************
*   GET_INT
*******************************************************************************
           DATA                                *** DATA SECTION ***
data_start  equ         *                      DB will point here
* External Entry Prologue (XEP)                Callers enter here
get_int     lea         data_start,a0          Load data section address
jmp.l       get_int$proc        Jump to pure code

            extern.p    vfmt_$ws          Declare external routine
a$vfmt_$ws  ac          vfmt_$ws          Address constant for external routine

            PROC                              ***** PROCEDURE SECTION *****

get_int$proc procedure  'get_int',#0,d2/db
            move.l      a0,db             Set DB pointer

            move.l      8(sb),a0          Get arg = address of start of string
            clr.l       d0                Initialize result

 Loop       clr.l       d1                Get next character
            move.b      (a0)+,d1
            sub.b       #'0',d1           Convert digit to binary
            blt.s       Done              If result is negative..
            cmp.b       #9,d1             ..or greater than 9..
            bgt.s       Done              ..then character wasn't a digit
            add.l       d0,d0              Multiply result so far by 10...
            bvs.s       Overflow          (check for overflow at each step)
            move.l      d0,d2             temp = result*2
            asl.l       #2,d0             result*8
            bvs.s       Overflow
            add.l       d2,d0             result*10
            bvs.s       Overflow
            add.l       d1,d0             Add in new digit
            bvc.s       Loop              Back for next character, unless...

* Result overflowed! Output a message to errout stream and return -1.
* Call vfmt_$ws(stream_$errout, '*** (get_int) overflow%');

Overflow    pea         errmsg                Push addr of error message string
            pea         errout                Push addr of stream id
            move.l      a$vfmt_$ws,a0         Get address of vfmt_$ws
            jsr         (a0)                  Call it
            addq.l      #8,sp                 Pop args
            move.l      #-1,d0                Return -1 result

Done        return      get_int$proc          Exit

 * Constant Data

errout      dc.w        3                      stream_$errout
errmsg      da.b        '*** (get_int) overflow%.'
            END
```

EXAMPLE 4 illustrates setting a cleanup handler.

```
            MODULE      example_4
            entry.p     good_addr
*******************************************************************************
*   function good_addr (in va: univ_ptr): boolean; options(val_param)
*
*   This function checks the validity of an address, by attempting to access
*   it and seeing if it faults.
*******************************************************************************
true        equ         $FF                 Boolean true
false       equ         0                   Boolean false
pfm_$cleanup_set  equ   $03040003           Status code

            DATA                            *** DATA SECTION ***
data_start  equ         *

good_addr   lea         data_start,a0       XEP
            jmp.l       good_addr$proc
cl_rec      ds.l        16                  Cleanup record
stat        ds.l        1                   Status code

           extern.p    pfm_$cleanup        External references
a$pfm_$cleanup  ac       pfm_$cleanup
           extern.p    pfm_$rls_cleanup
a$pfm_$rls_cleanup  ac  pfm_$rls_cleanup
           extern.p    pfm_$enable
a$pfm_$enable  ac       pfm_$enable

           PROC                             ***** PROCEDURE SECTION *****
*
* Note that we save all preservable registers
*
good_addr$proc procedure 'good_addr',#0,a2-a5/d2-d7

            move.l      a0,db               Set DB pointer
*
* Set a cleanup handler. (Caution: pfm_$cleanup changes SP, so don't make any
* SP-relative references after calling it.)
*
            pea         cl_rec              stat := pfm_$cleanup(cl_rec)
            move.l      a$pfm_$cleanup,a0
            jsr         (a0)
            add.w       #4,sp
*
* Test returned cleanup status
*
            cmp.l       #pfm_$cleanup_set,d0
            bne         Faulted
*
* Access given address and see if we fault
*
            move.l      8(sb),a0            Get address
            tst.b       (a0)                Access it
*
* If get here there was no fault - address is valid.  Release cleanup handler
* and return true.
```

Continued

Continued

```
            pea         stat                pfm_$rls_cleanup(cl_rec, stat)
            pea         cl_rec
            move.l      a$pfm_$rls_cleanup,a0
            jsr         (a0)
            add.w       #8,sp
            move.l      #true,d0            result = true
            bra.s       Done
*
* Cleanup handler invoked from fault - address is not valid.  Re-enable
* faults and return false.  (Actually, we should check that it's not a quit
* fault or something.)
* Note that all registers except DB, SB, & SP may be changed when we get
* here.
*
Faulted     move.l      a$pfm_$enable,a0    pfm_$enable()
            jsr         (a0)
            move.l      #false,d0           result = false
*
Done        return      good_addr$proc      Exit (restoring all registers)


            END
```

EXAMPLE 5 illustrates the use of floating-point registers.

```
                MODULE      example_5
                cpu         68020,68881
                entry.p     norm_rand

*******************************************************************************
*  function norm_rand ( in mean, std_dev : real ): real; options(val_param);
*
*   Returns a random variable drawn from an approximately normal distribution
*   with given mean and standard deviation.  Algorithm: sum 12 uniform 0-1
*   random variables to get an approximately normal variable with mean = 6 and
*   standard deviation = 1, then adjust to desired mean and std. dev.  Exter-
*   nal function 'rand' returns uniform 0-1 random variables.
*
*
*   Stack Frame:
*
*             +---------------+
*      SP-->| saved FP regs | -24 (offset from SB)
*           | (1 @ 12 bytes)|
*           |               |
*           +---------------+
*           | saved regs    | -12
*           | (3 @ 4 bytes) |
*           |               |
*           +---------------+
*      SB-->| link          |  0
*           +---------------+
*           | FCB ptr       | +4
*           +---------------+
*           | return addr   | +8
*           +---------------+
*           | mean          | +12
*           +---------------+
*           | std_dev       | +16
*           +---------------+
*******************************************************************************

* Offset of constant zero in 68881 rom (for fmovecr instruction)
 fpzero     equ         #15

* Define a dummy block representing the lower part of the stack frame.
            defs        (sb)
 link       ds.l        1
 fcb_ptr    ds.l        1
 ret_addr   ds.l        1
 mean       ds.l        1
 std_dev    ds.l        1
            ends
* Define a dummy block representing the upper part of the stack frame.
* Note that this is in backwards order.
            defds       (sb)
 save_ad    ds.l        3                   saved registers (d2, a2, db)
 save_fp    ds.l        3                   saved FP registers (fp2)
            ends
```

Continued

Continued

```
            DATA                                *** DATA SECTION ***
data_start  equ         *

norm_rand   lea         data_start,a0           XEP
            jmp.l       norm_rand$proc
*
            extern      rand                    External references
a$rand      ac          rand


            PROC                                ***** PROCEDURE SECTION *****
*
* Frame control block for floating-point registers
*
fcb         dc.w        1                       Type = MC68881
            dc.w        $20                     Reg mask: fp2 saved
            dc.l        save_fp-link            Save area offset from (SB)
*
* Pure code (note that "procedure" doesn't handle FCBs, so need explicit
* prologue)
*
norm_rand$proc procedure 'norm_rand',nocode,standard
            pea         fcb+1                   Push FCB for saved fp registers
            link        sb,#0                   Link
            movem.l     d2/a2/db,-(sp)          Save registers
            fmovem.x    fp2,-(sp)               Save fp registers
            move.l      a0,db                   Set DB pointer
 *
 * Sum 12 uniform random variables. Note that loop variables are kept in
 * registers preserved by the call to rand.
 *
            move.l      #11,d2                  Loop counter (12 trips)
            fmovecr     fpzero,fp2              Sum
            move.l      a$rand,a2               Addr of 'rand'
 loop       jsr         (a2)                    Call rand (result in d0)
            fadd.s      d0,fp2                  Accumulate sum
            dbra        d2,loop                 Loop
 *
 * FP2 is now normal(6,1).  Adjust to desired mean and standard deviation.
 *
            fsub.w      #6,fp2                  normal(0,1)
            fsglmul.s   std_dev,fp2             normal(0,std_dev)
            fadd.s      mean,fp2                normal(mean,std_dev)
            fmove.s     fp2,d0                  Return result in D0
 *
 * Exit
 *
            fmovem.x    save_fp,fp2             Restore FP registers
            movem.l     save_ad,d2/a2/db        Restore A/D registers
            unlk        sb                      Pop frame
            add.l       #4,sp                   Pop FCB pointer
            rts                                 Exit

            END
```

Chapter 7

Mathematical Libraries

In this chapter, we discuss two software packages that are known as mathematical libraries:

- Integer Arithmetic Library
- Floating-Point Package

The first part of the chapter provides the integer arithmetic library functions in the form of a Pascal insert file. The second part of the chapter describes the Floating-Point Package.

7.1 Integer Arithmetic Library

The integer arithmetic library implements 32-bit multiplication, division, and related operations not supported by processor hardware.

All arguments are passed by value; the results are returned in register D0. Arguments are not aligned to long word boundaries. Cases for which there are no explicit functions (for example long**short) are usually handled by extending short arguments to long.

> NOTE: Although the MC68020 implements most of these functions in hardware, programs that use 68020-specific instructions are constrained to run on 68020-based workstations. Therefore, even on these workstations, it is often preferable to call the library routines rather than compile separate versions of the program. The library itself is processor-specific and exploits 68020 instructions where available, so the performance penalty is just the call overhead.

```
type unsigned32 = 0..2147483647;     { unsigned 32-bit integer }
     unsigned16 = 0..65535;            { unsigned 16-bit integer }
```

7.1.1 Multiplication

This section lists the multiplication functions.

```
function m$mis$lll (x: integer32; y: integer32): integer32;      { x * y }
    val_param; extern;

function m$mis$llw (x: integer32; y: integer16): integer32;      { x * y }
    val_param; extern;

function m$miu$lll (x: unsigned32; y: unsigned32): unsigned32;   { x * y }
    val_param; extern;

function m$miu$llw (x: unsigned32; y: unsigned16): unsigned32;   { x * y }
    val_param; extern;
```

7.1.2 Division

This section lists the division functions.

```
function m$dis$lll (x: integer32; y: integer32): integer32;      { x / y }
    val_param; extern;

function m$dis$wwl (x: integer16; y: integer32): integer16;      { x / y }
    val_param; extern;

function m$dis$llw (x: integer32; y: integer16): integer32;      { x / y }
    val_param; extern;

function m$diu$lll (x: unsigned32; y: unsigned32): unsigned32;   { x / y }
    val_param; extern;

function m$diu$llw (x: unsigned32; y: unsigned16): unsigned32;   { x / y }
    val_param; extern;
```

7.1.3 Modulus

This section lists the modulus functions.

NOTE: The following functions deal with negative arguments using the rule

```
x mod y = sign(x) * (abs(x) mod abs(y))
```

where sign(x) = −1 if x is negative and +1 if x is positive. Example: −10 mod 3 = −1.

```
function m$oiss$lll (x: integer32; y: integer32): integer32;     { x mod y }
    val_param; extern;
```

```
function m$ois$wlw (x: integer32; y: integer16): integer16;      { x mod y }
    val_param; extern;

function m$ois$wwl (x: integer16; y: integer32): integer16;      { x mod y }
    val_param; extern;

function m$oiu$lll (x: unsigned32; y: unsigned32): unsigned32;   { x mod y }
    val_param; extern;

function m$oiu$wlw (x: unsigned32; y: unsigned16): unsigned16;   { x mod y }
    val_param; extern;
```

NOTE: The following functions use the ISO Pascal definition of mod, namely

```
x mod y  = x - floor(x/y)*y   for y > 0

         = undefined          for y < 0
```

where floor(t) = largest integer less than or equal to t. This differs from the above functions for negative arguments. Example: -10 mod 3 = 2.

```
function m$iis$lll (x: integer32; y: integer32): integer32;      { x mod y }
    val_param; extern;

function m$iis$wlw (x: integer32; y: integer16): integer16;      { x mod y }
    val_param; extern;

function m$iis$wwl (x: integer16; y: integer32): integer16;      { x mod y }
    val_param; extern;
```

7.1.4 Exponentiation

This section lists the exponentiation functions.

```
function m$eis$lll (x: integer32; y: integer32): integer32;      { x ** y }
    val_param; extern;

function m$eis$www (x: integer16; y: integer16): integer16;      { x ** y }
    val_param; extern;
```

7.2 Floating-Point Package (FPP)

The Floating-Point Package (**FPP**) provides a consistent interface to floating-point arithmetic that enables floating-point programs to run on any DOMAIN node. The FPP, which is a part of the global library SYSLIB, uses different implementations of the same interface depending on the hardware. At system startup, the system loads the appropriate version of SYSLIB for the model of your node. When you make calls to the FPP library from your DOMAIN assembly language program, the system makes use of the available FP hardware.

You can also use **in-line floating-point (FP) instructions** within your DOMAIN assembly language program instead of using FPP, if your node has either a PEB, 68881, or TERN (processor for DN460, DN660, and DSP160) hardware. In-line FP instructions enables your programs to run faster but restricts

them to execute on machines with the appropriate hardware. For detailed information on the 68881 floating-point instruction set, refer to the Motorola 68881 manual. For a list of TERN instructions, refer to Appendix C.

7.2.1 FPP Implementations

The FPP has five implementations that enable you to use the appropriate package for taking advantage of your hardware configuration. The five implementations are: *software, PEB, TERN, 68020,* and *68881.* We illustrate these FPP implementations in Figure 7-1.

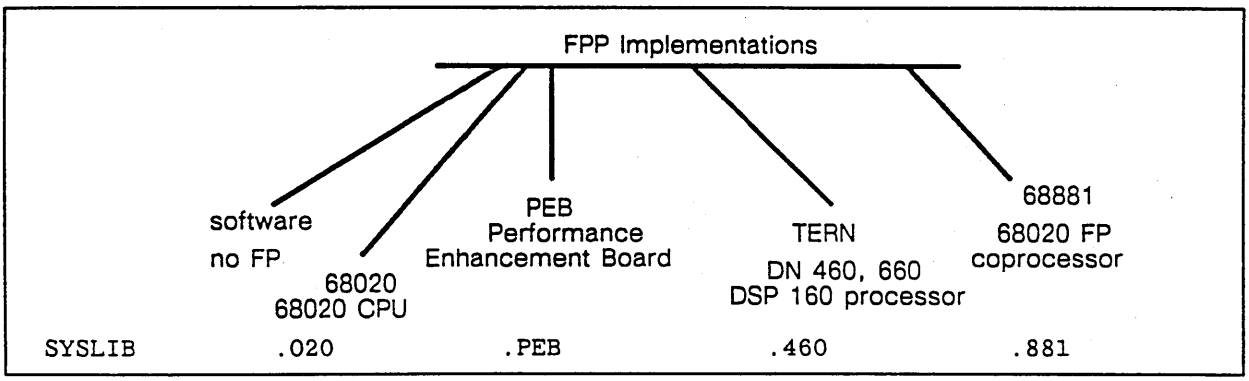


Figure 7-1. FPP Implementations and SYSLIB Extension Names

To see how compilers generate code that uses FPP to perform floating-point operations, use the **-CPU ANY** option in the command line when compiling a high-level language program containing floating-point operations. For example,

```
$ FTN float_test.ftn  -CPU ANY -EXP
```

The **-CPU ANY** options instructs the compiler to generate calls to FPP in order to perform floating-point operations. The **-EXP** options gives you an expanded listing file (.lst) in which you can see how the compiler generates code to use FPP.

7.2.2 FPP Library Calling and Exiting Conventions

The FPP model is a single accumulator machine. Depending on the node hardware, such as PEB, TERN, FP software, etc., the floating accumulator (FAC) resides at various locations. FAC is not preserved across calls to FPP.

In general, because FPP is a single accumulator, when FPP encounters two operands, such as with floating-point add (FPP_$SAV), it performs the next FPP operation in the instruction stream using the accumulator and the operand, that is, on top of the stack. Then, FPP places the result in the accumulator before popping the next operand off the stack and advancing to the next FPP operation in the instruction stream.

Within this section, we present a example of how to use FPP with the equation, `A = B + C`, where `B = .5` and `C = 16.2`.

Calling FPP

To call FPP from your DOMAIN assembly language program, perform the following steps:

1. Declare the appropriate external(s) before referencing FPP entry points.

```
EXTERN.P        FPP_$
EXTERN.P        FPP_$ESLV
EXTERN.P        FPP_$ESLA
EXTERN.P        FPP_$ESLC
```

   The four FPP entry points shown above are: `FPP_$`, `FPP_$ESLV` (Single Precision Load Value), `FPP_$ESLA` (Single Precision Load Address), and `FPP_$ESLC` (Single Precision Load Constant). The last three entry points load the FAC with a single precision value. To use FPP for DP operations, use the `FPP_$` entry point.

2. Push addresses and values of operands onto the stack in reverse order of their use.

```
PEA             A
MOVE.L          B,-(SP)
MOVE.L          C,-(SP)
```

3. Call FPP. Note here that JSR does not return from the FPP subroutine *until after* an FPP exit operation. Refer to the "Exiting FPP" section within this chapter.

```
        MOVE.L          FP_START,A0
        JSR             (A0)                *Note that FPP_$ESLV causes FAC
*                                            to be set to the value of C.
```

4. List the words representing floating-point operations and constants. Exit FPP using either FPP_$EXIT or an equivalent operation.

```
        DC.W            FPP_$SAV    *Add the values of C and B
        DC.W            FPP_$SSTX   *Store the result of A and EXIT
```

5. Define the operands in the DATA section.

```
        DATA
A       DS.L            1
B       DC.L            $3F000000   *B = .5
C       DC.L            $4181999A   *C = 16.2
FP_START AC             FPP$_ESLV   *Note FPP_$ESLV declared in step 1.
```

Exiting FPP

FPP exits on `FPP_$EXIT` or several other operate and exit codes, such as compare and exit, store and exit, and convert to integer and exit. Several store operations can cause an exit from FPP. Some of the operations that exit from FPP set the condition codes properly for the result. For example, the floating test and compare operation (`FPP_$TSXT`) exits with the condition codes set and with the D0 register set to a value that reflects the condition of the result (- , 0, +).

you call the FPP, it destroys all register contents except contents of
7 and A2–A6. Also, the contents of D6 and D7 are destroyed if FPP does
omplex operations, such as FPP_$CCA. However, on a DNx60 0r 68881,
preserve FP2 ... FP7. FPP does not follow the standard calling conven-
descrit n Chapter 6.

ting-Point Operations

listing from the FPP library insert file. Use these operations (separately or in com-
ne FPP. The table contains all the floating-point operations, shows the operation's
l representation, and provides an explanation for each operation.

Table 7-1 FPP Floating-Point Operations

	Meaning
t operations:	
ting ACcumulator)	
ble Precision)	
operations are Single Precision S.P. unless D.P. is specified)	
$FF34	st S.P. or D.P. FAC for -,0,+; set cc's & D0.L then exit
$FF3E	Same as FPP_$SSTA but exits when done (sets cc's)
$FF62	Return to caller beginning with instruction in next word
$FF6A	FAC := (SP)+ (Single precision Load Value)
$FF8A	FAC := ((SP)+) (Single precision Load using Address)
$FFAC	FAC := Next four bytes in instruction stream (Constant)
$FFD0	((SP)+) := FAC (Single precision STore using Addr)
$0000	FAC := FAC + (SP)+ (Single precision Add Value)
$0004	FAC := FAC + ((SP)+)
$0008	FAC := FAC + Next four bytes (Constant)
$000C	FAC := FAC - (SP)+ (Single precision Subtract Value)
$0010	FAC := FAC - ((SP)+)
$0014	FAC := FAC - Next four bytes (Constant)
$0018	FAC := (SP)+ - FAC (Single precision Inverse Subtract Value)
$001C	FAC := ((SP)+) - FAC
$0020	FAC := Next four bytes (Constant) - FAC
$0024	FAC := FAC * (SP)+ (Single precision Multiply Value)
$0028	FAC := FAC * ((SP)+)
$002C	FAC := FAC * Next four bytes (Constant)
$0030	FAC := FAC / (SP)+ (Single precision Divide Value)
$0034	FAC := FAC / ((SP)+)
$0038	FAC := FAC / Next four bytes (Constant)

Continued on next page

Operations	Hex	Meaning

```
FPP_$SSQR  equ $00DC Take square root of FAC
FPP_$DSQR  equ $00E0 Take square root of DFAC
fpp_$sexp  equ $00E4 EXP(<FAC>)
fpp_$dexp  equ $00E8 DEXP(<DFAC>)
fpp_$slog  equ $00EC ALOG(<FAC>)
fpp_$dlog  equ $00F0 DLOG(<DFAC>)
fpp_$ssin  equ $00F4 SIN(<FAC>)
fpp_$dsin  equ $00F8 DSIN(<DFAC>)
fpp_$scos  equ $00FC COS(<FAC>)
fpp_$dcos  equ $0100 DCOS(<DFAC>)
fpp_$stan  equ $0104 TAN(<FAC>)
fpp_$dtan  equ $0108 DTAN(<DFAC>)
fpp_$satan equ $010C ATAN(<FAC>)
fpp_$datan equ $0110 DATAN(<DFAC>)
fpp_$satan2a  equ $0114 ATAN2(<FAC>,((sp+))
fpp_$satan2v  equ $0118 ATAN2(<FAC>,(sp+)
fpp_$satan2c  equ $011C ATAN2(<FAC>,<CONST>)
fpp_$datan2a  equ $0120 DATAN2(<DFAC>,((sp+))
fpp_$datan2c  equ $0124 DATAN2(<DFAC>,<CONST>)
fpp_$e$21v equ $0128 E$21(<FAC>,(sp)+)
fpp_$e$22a equ $012C E$22(<FAC>,((sp)+))
fpp_$e$22v equ $0130 E$22(<FAC>,(sp)+)
fpp_$e$22c equ $0134 E$22(<FAC>,<CONST>)
fpp_$e$61v equ $0138 E$61(<DFAC>,(sp)+)
fpp_$e$62a equ $013C E$62(<DFAC>,((sp)+))
fpp_$e$62v equ $0140 E$62(<DFAC>,(sp)+)
fpp_$e$62c equ $0144 E$62(<DFAC>,<CONST>)
fpp_$e$66a equ $0148 E$66(<DFAC>,((sp)+))
fpp_$e$66c equ $014C E$66(<DFAC>,<CONST>)
FPP_$STRUNC equ $0150 FAC := Int[FAC]
FPP_$SNINT  equ $0154 FAC := Nint[FAC] (Nearest integer)
FPP_$DTRUNC equ $0158 D.P. FAC := Int[FAC]
FPP_$DNINT  equ $015C D.P. FAC := Nint[FAC] (Nearest integer)
FPP_$SMINV EQU $0160 FAC := Min[FAC, (SP)+]
FPP_$SMINA EQU $0164 FAC := Min[FAC, ((SP)+)]
FPP_$SMINC EQU $0168 FAC := Min[FAC, Next 4 bytes]
FPP_$SMAXV EQU $016C FAC := Max[FAC, (SP)+]
FPP_$SMAXA EQU $0170 FAC := Max[FAC, ((SP)+)]
FPP_$SMAXC EQU $0174 FAC := Max[FAC, Next 4 Bytes]
FPP_$DMINA EQU $0178 D.P. FAC := Min[FAC, ((SP)+)]
FPP_$DMINC EQU $017C D.P. FAC := Min[FAC, Next 8 bytes]
FPP_$DMAXA EQU $0180 D.P. FAC := Max[FAC, ((SP)+)]
FPP_$DMAXC EQU $0184 D.P. FAC := Max[FAC, Next 8 bytes]
FPP_$CLA   EQU $0188 Complex FAC := ((SP)+)
FPP_$CLC   EQU $018C Complex FAC := Next 8 bytes (Constant)
```

Continued on next page

Operations	Hex	Meaning

```
FPP_$CAA    EQU $0190 Complex FAC := FAC + ((SP)+)
FPP_$CAC    EQU $0194 Complex FAC := FAC + Next 8 bytes
FPP_$CSA    EQU $0198 Complex FAC := FAC - ((SP)+)
FPP_$CSC    EQU $019C Complex FAC := FAC - Next 8 bytes
FPP_$CISA   EQU $01A0 Complex FAC := ((SP)+) - FAC
FPP_$CISC   EQU $01A4 Complex FAC := Next 8 bytes - FAC
FPP_$CMA    EQU $01A8 Complex FAC := FAC * ((SP)+)
FPP_$CMC    EQU $01AC Complex FAC := FAC * Next 8 bytes
FPP_$CDA    EQU $01B0 Complex FAC := FAC / ((SP)+)
FPP_$CDC    EQU $01B4 Complex FAC := FAC / Next 8 bytes
FPP_$CIDA   EQU $01B8 Complex FAC := ((SP)+) / FAC
FPP_$CIDC   EQU $01BC Complex FAC := Next 8 bytes / FAC
FPP_$CSTA   EQU $01C0 Store Complex FAC thru (SP)+
FPP_$CSTX   EQU $01C4 Store Complex FAC thru (SP)+ and exit
FPP_$CSWAP  EQU $01C8 Exchange real and imaginary parts of FAC
FPP_$CCNV   EQU $01CC Convert Single Prec to Complex (Set the imag part to 0)
FPP_$CCONJ  EQU $01D0 Calc Complex Conjugate (negate imag part of FAC)
FPP_$SLUV   EQU $01D4 FAC := Float[(SP)+]       (Float unsigned 32-bit integer)
FPP_$DLUV   EQU $01D8 D.P. FAC := Float[(SP)+]  (DP fl unsigned 32-bit integer)
FPP_$CCVX   EQU $0028 Complex Compare FAC : (SP)+; set cc's & D0.L then exit
FPP_$CCAX   EQU $002c Complex Comare FAC : ((SP)+); set cc's & D0.L then exit
FPP_$CCCX   EQU $0230 Complex Compare FAC : Next 8 bytes; set cc's & D0.L then
                      exit
```

7.2.4 Notes On FPP

The following are important notes on FPP.

- FAC can contain either Single Precision (SP) or Double Precision (DP) values, but not both. Use these conversion operations to change between the types:

FPP_$SCNV	Converts DP FAC to SP.
FPP_$DCNV	Converts SP FAC to DP.

- Normalize all floating-point operands. The 68881 version of FPP does not return all floating-point numbers normalized. In many cases, certain operations can result in denormalized numbers. **Denormalized numbers** occur when the numbers do not fit within the dynamic range of the precision in which you are working, but are not so far out of range that an overflow or underflow occurs.

- Zero is represented as either positive or negative zero. To check for 0.0, look at all but the high bit of the number.

Appendixes

Appendix A

Error Codes and Messages

This appendix contains a list of error codes and messages. In addition, the appendix provides an explanation for many of the error messages. Currently, the error code does not appear with the displayed error message.

Error Code	Message	Meaning
E1	'Assembler Error'	Internal bug. Please submit a UCR.
E2	'Symbol Already Defined'	You defined the symbol more than once in the program.
E3	'Illegal Symbol'	The symbol you used in the specified field was not valid. Refer to *Special Characters Table* in Chapter 3.
E4	'Illegal Suffix'	The suffix you used was illegal for the instruction. For example, **CHK.L Capr,D0** (legal on 68020). Refer to Appendix B for a list of legal suffixes.
E5	'Value Out Of Range'	The value you chose was invalid for the instruction. For example, the value of 16 is out of range for the following link instruction: **link sb, #16.** Refer to the appropriate instruction set for valid ranges.

	Meaning
ress'	The address you used in the specified field was not valid. You may have used a data register instead of an address register.
ıe'	The addressing mode you used was not valid for the instruction. For example, you may have used **6-(a0)** instead of **6(a0)**. Refer to Chapter 3 for more information.
thesis Expected'	You forgot the right parenthesis.
Symbol'	The symbol or variable you used was not defined in the program. Perhaps you misspelled the symbol name.
x Size Suffix'	The suffix size was illegal for the index in the specified field. For example: **CLR 0(A0,D0.X)** where .X is not allowed.
Register'	(obsolete)
x Register'	(obsolete)
ed Instruction'	The instruction in the specified field was not valid for the instruction set. Consult Appendix B for a list of valid instructions.
ix for Instruction'	The suffix you used with the instruction is not valid for the instruction set. Consult Appendix B for the valid suffixes.
ol in Address Field'	The symbol you used in the address field was not valid. Refer to the *Special Characters Table* in Chapter 3.
d Required'	You forgot to specify any data after the instruction.
Field Required'	You omitted the data required by the destination field in the instruction. Refer to the "Instruction Format" section in Chapter 3.

Error Code	Message	Meaning
E51	'Register Not in Use'	There is no corresponding **USING** pseudo-op for the register in the variable field of the **DROP** pseudo-op.
E52	'Register Already In Use'	A **DROP** pseudo-op must appear between **USING** pseudo-ops with the same register. Also, note that there is an implicit **USING** of **A5** at the start of every module.
E53	'Multiple Program or Module Stmt'	Only one **PROGRAM** or **MODULE** pseudo-op may appear in an assembly language module.
E54	'Unbalanced Conditional Processor Directive'	**%ELSEIF**, **%ELSE**, or **%ENDIF** directive encountered with no matching **%IF** directive. Refer to the "Conditional Assembly" section in Chapter 4.
E55	'Conditional Processor Warning'	**%WARNING** directive encountered. Refer to the "Conditional Assembly" section in Chapter 4.
E56	'Conditional Processor Error'	**%ERROR** directive encountered. Refer to the "Conditional Assembly" section in Chapter 4.
E57	'Conditional Processor Directive Syntax Error'	Refer to the "Conditional Assembly" section in Chapter 4.
E58	'Data Or Address Register Required'	Data or address register required an instruction operand. Refer to the instruction description for more information.
E59	'Floating-Point Register Required'	Floating-point register required an instruction operand. Refer to the description of the instruction for more information.
E60	'Memory Alterable Address Required'	Instruction requires memory alterable operand. Refer to the description of the instruction for more information.

Error Code	Message	Meaning
E61	'FMOVEM Requires Register List Or Data Reg'	**FMOVEM** instruction requires floating-point register list or data register containing dynamic bit mask. Refer to the description of the instruction for more information.
E62	'Relocatable Immediate Requires Long Word Instruction'	Op-code of the instruction that contains relocatable immediate operand must have implicit size of long or explicit .L extension.
E63	'Data Address Other Than Immediate Required'	An immediate is not allowed as the destination of an instruction. Refer to the instruction description for more information.
E64	'Decimal Value Not Supported For Floating- Point'	A floating-point op-code with a extension of **.D**, **.S**, or **.X** cannot have an immediate operand that is a decimal number. FPP currently does not support floating point numbers; only hex numbers are currently allowed.
E65	'Illegal Suffix For Floating-Point FPr, FPr Instruction'	The op-code on a floating-point register to register instruction cannot have a extension.
E66	'Right Bracket Expected'	You forgot the right bracket.
E67	'Illegal Outer Displacement Value'	(obsolete)
E68	'Illegal Bitfield Value'	The bitfield value exceeded the maximum bitfield width of 31 in the specified field. For example, 47 exceeds the maximum: **BFCHG (A0){D1:47} *<ea>{offset:width}**.
E69	'Only One Level Of Indirection Allowed'	You used more than one level of indirection. Indirection involves the use of [ ].
E70	'Bitfield Specification Required'	You omitted the bitfield in the specified field. For example, **BFCLR BIT_LESS.**
E71	'Bitfield Spec Not Allowed'	You used a bitfield specification { } where it is not allowed.

Error Code	Message	Meaning
E72	'Improper Register Pair'	The register pair you used was invalid. For example, **DIVSL kym,d0:kym** is invalid. **DIVSL kym,d0:d1** is valid.
E73	'Register Pair Required'	You need to use a register pair in the specified field. For example, you used **DIVSL foo,D0** instead of **foo,D0:D1.**
E74	'Control Register Required'	You need to use a control register in the specified field. For example, **MOVEC OUT,D0** should be **MOVEC #OUT,D0.**
E75	'Mask Value Required'	The argument for **PFLUSH** is not absolute. This is a 68851 error.
E76	'Bad Argument'	The level is not absolute on **PTEST.** This is a 68851 error.
E77	'Read Or Write Must Be Specified'	Missing READ/WRITE argument on **PLOAD.** This is a 68851 error.
E78	'Level Field Required'	Missing level argument on **PTEST.** This is a 68851 error.
E79	'Nostack Requires Nocode And Precludes Standard, #-N, Register List Arguments'	The NOSTACK argument for **PROCEDURE** must be accompanied by NOCODE and cannot be used with STANDARD, #-N, or register list arguments.

Appendix B

Legal Op-code and Pseudo-Op Mnemonics

This appendix contains information regarding the legal usages of the DOMAIN assembly language instruction set. We provide the instruction name, the list of valid machine types on which the instruction is valid, and the legal extensions (suffixes).

To use this appendix, simply look up the instruction. For example, the **ADD** instruction is valid on machines containing either the MC68000, MC68010, or MC68020 microprocessor. **ADDQ**, **ADD.B**, **ADD.W**, and **ADD.L** are all legal suffixes of the **ADD** instruction. Note that pseudo-ops are listed in **BOLDFACE TYPE**. For complete information on any of the instructions below, consult the Motorola instruction set in the appropriate Motorola manual. For additional information on pseudo-ops, refer to Chapter 4 in this manual.

We divide this appendix into the following topics:

- Valid Machine Types
- Legal Suffixes
- Legal Op-code and Pseudo-op Mnemonics

B.1 Valid Machine Types

The following are valid machine types for DOMAIN assembly language. Refer to the appropriate Motorola manual for information on the Motorola instruction set.

```
68851 (PMMU) INSTRUCTION -- Motorola
68881 (FPPU) INSTRUCTION -- Motorola
TERN  INSTRUCTION -- DOMAIN
68000 INSTRUCTION -- Motorola
68010 INSTRUCTION -- Motorola
68020 INSTRUCTION -- Motorola
```

B.2 Legal Suffixes

The following is a list of legal suffixes for DOMAIN assembly language.

```
        No suffix for example, BRA means use 16-bit displacement.
 Q      Quick. For example,SUBQ, ADDQ
.B      8 bit byte
.S      Short. For example, BRA.S means use  8-bit displacement.
.W      16 bit word
.L      32 bit long word
.D      legal
.X      legal
```

B.3 Legal Op-code and Pseudo-op Mnemonics

The following is a list of legal op-code and pseudo-op mnemonics for DOMAIN assembly.

Instruction	Valid Machine Types			Legal Suffixes
ABCD	68000	68010	68020	.B
AC	68000	68010	68020	.L
ADD	68000	68010	68020	Q .B .W .L
ADDA	68000	68010	68020	.W .L
ADDI	68000	68010	68020	.B .W .L
ADDQ	68000	68010	68020	.B .W .L
ADDX	68000	68010	68020	.B .W .L
AND	68000	68010	68020	.B .W .L
ANDI	68000	68010	68020	.B .W .L
ASL	68000	68010	68020	.B .W .L
ASR	68000	68010	68020	.B .W .L
BCC	68000	68010	68020	.S .B .W .L
BCHG	68000	68010	68020	.B .L
BCLR	68000	68010	68020	.B .L
BCS	68000	68010	68020	.S .B .W .L
BEQ	68000	68010	68020	.S .B .W .L
BFCHG	68020			
BFCLR	68020			
BFEXTS	68020			

Instruction	Valid Machine Types			Legal Suffixes
BFEXTU	68020			
BFFFO	68020			
BFINS	68020			
BFSET	68020			
BFTST	68020			
BGE	68000	68010	68020	.S .B .W .L
BGT	68000	68010	68020	.S .B .W .L
BHI	68000	68010	68020	.S .B .W .L
BKPT	68020			
BLE	68000	68010	68020	.S .B .W .L
BLS	68000	68010	68020	.S .B .W .L
BLT	68000	68010	68020	.S .B .W .L
BMI	68000	68010	68020	.S .B .W .L
BNE	68000	68010	68020	.S .B .W .L
BPL	68000	68010	68020	.S .B .W .L
BRA	68000	68010	68020	.S .B .W .L
BSET	68000	68010	68020	.B .L
BSR	68000	68010	68020	.S .B .W .L
BTST	68000	68010	68020	.B .L
BVC	68000	68010	68020	.S .B .W .L
BVS	68000	68010	68020	.S .B .W .L
CALLM	68020			
CAS	68020			.B .W .L
CHK	68000 (.W only)		68020	.W .L
CHK2	68020			.B .W .L
CLR	68000	68010	68020	.B .W .L
CMP	68000	68010	68020	.B .W .L

Instruction	Valid Machine Types			Legal Suffixes
CMP2	68020			.B .W .L
CMPA	68000	68010	68020	.W .L
CMPI	68000	68010	68020	.B .W .L
CMPM	68000	68010	68020	.B .W .L
CPU	68000	68010	68020	
DA	68000	68010	68020	.B .W .L
DATA	68000	68010	68020	
DBCC	68000	68010	68020	
DBCS	68000	68010	68020	
DBEQ	68000	68010	68020	
DBF	68000	68010	68020	
DBGE	68000	68010	68020	
DBGT	68000	68010	68020	
DBHI	68000	68010	68020	
DBLE	68000	68010	68020	
DBLS	68000	68010	68020	
DBLT	68000	68010	68020	
DBMI	68000	68010	68020	
DBNE	68000	68010	68020	
DBPL	68000	68010	68020	
DBRA	68000	68010	68020	
DBT	68000	68010	68020	
DBVC	68000	68010	68020	
DBVS	68000	68010	68020	
DC	68000	68010	68020	.B .W .L
DCNT	68000	68010	68020	
DEFDS	68000	68010	68020	

Instruction	Valid Machine Types				Legal Suffixes
DEFS	68000	68010	68020		
DFSECT	68000	68010	68020		
DIV32S	DNx60 TERN				.L
DIV32U	DNx60 TERN				.L
DIVS	68000	68010	68020		.W .L
DIVSL	68020				.L
DIVU	68000	68010	68020		.W .L
DIVUL	68020				.L
DROP	68000	68010	68020		
DS	68000	68010	68020		.B .W .L
EJECT	68000	68010	68020		
ELSE	68000	68010	68020		
END	68000	68010	68020		
ENDS	68000	68010	68020		
ENTRY	68000	68010	68020		.B .W .L
EOR	68000	68010	68020		.B .W .L
EORI	68000	68010	68020		.B .W .L
EQU	68000	68010	68020		
EXG	68000	68010	68020		.L
EXT	68000	68010	68020		.W .L
EXTB	68020				.L
EXTERN	68000	68010	68020		.B .W .L
FABS	Valid with 68881		68020	68881	.S .B .W .L .D .X
FACOS	"	"	68020	68881	.S .B .W .L .D .X
FADD	"	"	68020	68881	.S .B .W .L .D .X
FASIN	"	"	68020	68881	.S .B .W .L .D .X
FATAN	68000	68010	68020	68881	.S .B .W .L .D .X

Instruction	Valid Machine Types				Legal Suffixes
FATANH	68000	68010	68020	68881	.S .B .W .L .D .X
FBEQ	68000	68010	68020	68881	.S
FBF	68000	68010	68020	68881	.S
FBGE	68000	68010	68020	68881	.S
FBGL	68000	68010	68020	68881	.S
FBGLE	68000	68010	68020	68881	.S
FBGT	68000	68010	68020	68881	.S
FBLE	68000	68010	68020	68881	.S
FBLT	68000	68010	68020	68881	.S
FBNEQ	68000	68010	68020	68881	.S
FBNGE	68000	68010	68020	68881	.S
FBNGL	68000	68010	68020	68881	.S
FBNGLE	68000	68010	68020	68881	.S
FBNGT	68000	68010	68020	68881	.S
FBNLE	68000	68010	68020	68881	.S
FBNLT	68000	68010	68020	68881	.S
FBOGE	68000	68010	68020	68881	.S
FBOGL	68000	68010	68020	68881	.S
FBOGT	68000	68010	68020	68881	.S
FBOLE	68000	68010	68020	68881	.S
FBOLT	68000	68010	68020	68881	.S
FBOR	68000	68010	68020	68881	.S
FBSEQ	68000	68010	68020	68881	.S
FBSF	68000	68010	68020	68881	.S
FBSNEQ	68000	68010	68020	68881	.S
FBST	68000	68010	68020	68881	.S
FBT	68000	68010	68020	68881	.S

Instruction	Valid Machine Types				Legal Suffixes
FBUEQ	68000	68010	68020	68881	.S
FBUGE	68000	68010	68020	68881	.S
FBUGT	68000	68010	68020	68881	.S
FBULE	68000	68010	68020	68881	.S
FBULT	68000	68010	68020	68881	.S
FBUN	68000	68010	68020	68881	.S
FCMP	68000	68010	68020	68881	.S .B .W .L .D .X
FCOS	68000	68010	68020	68881	.S .B .W .L .D .X
FCOSH	68000	68010	68020	68881	.S .B .W .L .D .X
FDBEQ	68000	68010	68020	68881	
FDBF	68000	68010	68020	68881	
FDBGE	68000	68010	68020	68881	
FDBGL	68000	68010	68020	68881	
FDBGLE	68000	68010	68020	68881	
FDBGT	68000	68010	68020	68881	
FDBLE	68000	68010	68020	68881	
FDBLT	68000	68010	68020	68881	
FDBNE	68000	68010	68020	68881	
FDBNGE	68000	68010	68020	68881	
FDBNGL	68000	68010	68020	68881	
FDBNGLE	68000	68010	68020	68881	
FDBNGT	68000	68010	68020	68881	
FDBNLE	68000	68010	68020	68881	
FDBNLT	68000	68010	68020	68881	
FDBOGE	68000	68010	68020	68881	
FDBOGL	68000	68010	68020	68881	
FDBOGT	68000	68010	68020	68881	

Instruction	Valid Machine Types				Legal Suffixes
FDBOLE	68000	68010	68020	68881	
FDBOLT	68000	68010	68020	68881	
FDBOR	68000	68010	68020	68881	
FDBSEQ	68000	68010	68020	68881	
FDBSF	68000	68010	68020	68881	
FDBSNEQ	68000	68010	68020	68881	
FDBST	68000	68010	68020	68881	
FDBT	68000	68010	68020	68881	
FDBUEQ	68000	68010	68020	68881	
FDBUGE	68000	68010	68020	68881	
FDBUGT	68000	68010	68020	68881	
FDBULE	68000	68010	68020	68881	
FDBULT	68000	68010	68020	68881	
FDBUN	68000	68010	68020	68881	
FDIV	68000	68010	68020	68881	.S .B .W .L .D .X
FETOX	68000	68010	68020	68881	.S .B .W .L .D .X
FETOXM1	68000	68010	68020	68881	.S .B .W .L .D .X
FGETEXP	68000	68010	68020	68881	.S .B .W .L .D .X
FGETMAN	68000	68010	68020	68881	.S .B .W .L .D .X
FINT	68000	68010	68020	68881	.S .B .W .L .D .X
FINTRZ	68000	68010	68020	68881	.S .B .W .L .D .X
FLOG10	68000	68010	68020	68881	.S .B .W .L .D .X
FLOG2	68000	68010	68020	68881	.S .B .W .L .D .X
FLOGN	68000	68010	68020	68881	.S .B .W .L .D .X
FLOGNP1	68000	68010	68020	68881	.S .B .W .L .D .X
FMOD	68000	68010	68020	68881	.S .B .W .L .D .X
FMOVE	68000	68010	68020	68881	.S .B .W .L .D .X

Instruction	Valid Machine Types				Legal Suffixes
FMOVECR	68000	68010	68020	68881	.X
FMOVEM	68000	68010	68020	68881	.X
FMUL	68000	68010	68020	68881	.S .B .W .L .D .X
FNEG	68000	68010	68020	68881	.S .B .W .L .D .X
FNOP	68000	68010	68020	68881	.S
FREM	68000	68010	68020	68881	.S .B .W .L .D .X
FRESTORE	68000	68010	68020	68881	
FSAVE	68000	68010	68020	68881	
FSCALE	68000	68010	68020	68881	.S .B .W .L .D .X
FSEQ	68000	68010	68020	68881	.B
FSF	68000	68010	68020	68881	.B
FSGE	68000	68010	68020	68881	.B
FSGL	68000	68010	68020	68881	.B
FSGLDIV	68000	68010	68020	68881	.S .B .W .L .D .X
FSGLE	68000	68010	68020	68881	.B
FSGLMUL	68000	68010	68020	68881	.S .B .W .L .D .X
FSGT	68000	68010	68020	68881	.B
FSIN	68000	68010	68020	68881	.S .B .W .L .D .X
FSINH	68000	68010	68020	68881	.S .B .W .L .D .X
FSLE	68000	68010	68020	68881	.B
FSLT	68000	68010	68020	68881	.B
FSNEQ	68000	68010	68020	68881	.B
FSNGE	68000	68010	68020	68881	.B
FSNGL	68000	68010	68020	68881	.B
FSNGLE	68000	68010	68020	68881	.B
FSNGT	68000	68010	68020	68881	.B
FSNLE	68000	68010	68020	68881	.B

Instruction	Valid Machine Types				Legal Suffixes
FSNLT	68000	68010	68020	68881	.B
FSOGE	68000	68010	68020	68881	.B
FSOGL	68000	68010	68020	68881	.B
FSOGT	68000	68010	68020	68881	.B
FSOLE	68000	68010	68020	68881	.B
FSOLT	68000	68010	68020	68881	.B
FSOR	68000	68010	68020	68881	.B
FSQRT	68000	68010	68020	68881	.S .B .W .L .D .X
FSSEQ	68000	68010	68020	68881	.B
FSSF	68000	68010	68020	68881	.B
FSSNEQ	68000	68010	68020	68881	.B
FSST	68000	68010	68020	68881	.B
FST	68000	68010	68020	68881	.B
FSUB	68000	68010	68020	68881	.S .B .W .L .D .X
FSUEQ	68000	68010	68020	68881	.B
FSUGE	68000	68010	68020	68881	.B
FSUGT	68000	68010	68020	68881	.B
FSULE	68000	68010	68020	68881	.B
FSULT	68000	68010	68020	68881	.B
FSUN	68000	68010	68020	68881	.B
FTAN	68000	68010	68020	68881	.S .B .W .L .D .X
FTANH	68000	68010	68020	68881	.S .B .W .L .D .X
FTENTOX	68000	68010	68020	68881	.S .B .W .L .D .X
FTEQ	68000	68010	68020	68881	
FTEST	68000	68010	68020	68881	.S .B .W .L .D .X
FTGE	68000	68010	68020	68881	
FTGL	68000	68010	68020	68881	

Instruction	Valid Machine Types				Legal Suffixes
FTGLE	68000	68010	68020	68881	
FTGT	68000	68010	68020	68881	
FTLE	68000	68010	68020	68881	
FTLT	68000	68010	68020	68881	
FTNE	68000	68010	68020	68881	
FTNGE	68000	68010	68020	68881	
FTNGL	68000	68010	68020	68881	
FTNGLE	68000	68010	68020	68881	
FTNGT	68000	68010	68020	68881	
FTNLE	68000	68010	68020	68881	
FTNLT	68000	68010	68020	68881	
FTOGE	68000	68010	68020	68881	
FTOGL	68000	68010	68020	68881	
FTOGT	68000	68010	68020	68881	
FTOLE	68000	68010	68020	68881	
FTOLT	68000	68010	68020	68881	
FTOR	68000	68010	68020	68881	
FTPEQ	68000	68010	68020	68881	.W .L
FTPGE	68000	68010	68020	68881	.W .L
FTPGL	68000	68010	68020	68881	.W .L
FTPGLE	68000	68010	68020	68881	.W .L
FTPGT	68000	68010	68020	68881	.W .L
FTPLE	68000	68010	68020	68881	.W .L
FTPLT	68000	68010	68020	68881	.W .L
FTPNE	68000	68010	68020	68881	.W .L
FTPNGE	68000	68010	68020	68881	.W .L
FTPNGL	68000	68010	68020	68881	.W .L

Instruction	Valid Machine Types				Legal Suffixes
FTPNGLE	68000	68010	68020	68881	.W .L
FTPNGT	68000	68010	68020	68881	.W .L
FTPNLE	68000	68010	68020	68881	.W .L
FTPNLT	68000	68010	68020	68881	.W .L
FTPOGE	68000	68010	68020	68881	.W .L
FTPOGL	68000	68010	68020	68881	.W .L
FTPOGT	68000	68010	68020	68881	.W .L
FTPOLE	68000	68010	68020	68881	.W .L
FTPOLT	68000	68010	68020	68881	.W .L
FTPOR	68000	68010	68020	68881	.W .L
FTPUEQ	68000	68010	68020	68881	.W .L
FTPUGE	68000	68010	68020	68881	.W .L
FTPUGT	68000	68010	68020	68881	.W .L
FTPULE	68000	68010	68020	68881	.W .L
FTPULT	68000	68010	68020	68881	.W .L
FTPUN	68000	68010	68020	68881	.W .L
FTST	68000	68010	68020	68881	.S .B .W .L .D .X
FTUEQ	68000	68010	68020	68881	
FTUGE	68000	68010	68020	68881	
FTUGT	68000	68010	68020	68881	
FTULE	68000	68010	68020	68881	
FTULT	68000	68010	68020	68881	
FTUN	68000	68010	68020	68881	
FTWOTOX	68000	68010	68020	68881	.S .B .W .L .D .X
ILLEGAL	68000	68010	68020		
JMP	68000	68010	68020		.S .L
JSR	68000	68010	68020		
LEA	68000	68010	68020		

Instruction	Valid Machine Types				Legal Suffixes
LINK	68000	68010	68020		.L
LIST	68000	68010	68020		
LSL	68000	68010	68020		.B .W .L
LSR	68000	68010	68020		.B .W .L
MODULE	68000	68010	68020		
MOVE	68000	68010	68020		.B .W .L
MOVEA	68000	68010	68020		.W .L
MOVEC	68010	68020			.L
MOVEM	68000	68010	68020		.W .L
MOVEP	68000	68010	68020		.W .L
MOVEQ	68000	68010	68020		.L
MOVES	68020				.B .W .L
MUL32S				TERN	.L
MUL32U				TERN	.L
MULS	68000	68010	68020		.W .L
MULU	68000	68010	68020		.W .L
NBCD			68020		.B
NEG	68000	68010	68020		.B .W .L
NEGX	68000	68010	68020		.B .W .L
NOLIST	68000	68010	68020		
NOP	68000	68010	68020		
NOT	68000	68010	68020		.B .W .L
OR	68000	68010	68020		.B .W .L
ORG	68000	68010	68020		
ORI	68000	68010	68020		.B .W .L
PACK	68020				
PBAC	68000	68010	68020	68851	.W .L
PBAS	68000	68010	68020	68851	.W .L

Instruction	Valid Machine Types				Legal Suffixes
PBBC	68000	68010	68020	68851	.W .L
PBBS	68000	68010	68020	68851	.W .L
PBCC	68000	68010	68020	68851	.W .L
PBCS	68000	68010	68020	68851	.W .L
PBGC	68000	68010	68020	68851	.W .L
PBGS	68000	68010	68020	68851	.W .L
PBIC	68000	68010	68020	68851	.W .L
PBIS	68000	68010	68020	68851	.W .L
PBLC	68000	68010	68020	68851	.W .L
PBLS	68000	68010	68020	68851	.W .L
PBSC	68000	68010	68020	68851	.W .L
PBSS	68000	68010	68020	68851	.W .L
PBWC	68000	68010	68020	68851	.W .L
PBWS	68000	68010	68020	68851	.W .L
PDBAC	68000	68010	68020	68851	
PDBAS	68000	68010	68020	68851	
PDBBC	68000	68010	68020	68851	
PDBBS	68000	68010	68020	68851	
PDBCC	68000	68010	68020	68851	
PDBCS	68000	68010	68020	68851	
PDBGC	68000	68010	68020	68851	
PDBGS	68000	68010	68020	68851	
PDBIC	68000	68010	68020	68851	
PDBIS	68000	68010	68020	68851	
PDBLC	68000	68010	68020	68851	
PDBLS	68000	68010	68020	68851	
PDBSC	68000	68010	68020	68851	
PDBSS	68000	68010	68020	68851	

Instruction	Valid Machine Types				Legal Suffixes
PDBWC	68000	68010	68020	68851	
PDBWS	68000	68010	68020	68851	
PEA	68000	68010	68020		
PFLUSH	68000	68010	68020	68851	
PFLUSHR	68000	68010	68020	68851	
PLOAD	68000	68010	68020	68851	
PMOVE	68000	68010	68020	68851	.B .W .L
PRESTORE	68000	68010	68020	68851	
PROC	68000	68010	68020		
PROGRAM	68000	68010	68020		
PSAC	68000	68010	68020	68851	.W .L
PSAS	68000	68010	68020	68851	.W .L
PSAVE	68000	68010	68020	68851	
PSBC	68000	68010	68020	68851	.W .L
PSBS	68000	68010	68020	68851	.W .L
PSCC	68000	68010	68020	68851	.W .L
PSCS	68000	68010	68020	68851	.W .L
PSGC	68000	68010	68020	68851	.W .L
PSGS	68000	68010	68020	68851	.W .L
PSIC	68000	68010	68020	68851	.W .L
PSIS	68000	68010	68020	68851	.W .L
PSLC	68000	68010	68020	68851	.W .L
PSLS	68000	68010	68020	68851	.W .L
PSSC	68000	68010	68020	68851	.W .L
PSSS	68000	68010	68020	68851	.W .L
PSWC	68000	68010	68020	68851	.W .L
PSWS	68000	68010	68020	68851	.W .L
PTEST	68000	68010	68020	68851	

Instruction	Valid Machine Types				Legal Suffixes
PTRAPAC	68000	68010	68020	68851	
PTRAPAS	68000	68010	68020	68851	
PTRAPBC	68000	68010	68020	68851	
PTRAPBS	68000	68010	68020	68851	
PTRAPCC	68000	68010	68020	68851	
PTRAPCS	68000	68010	68020	68851	
PTRAPGC	68000	68010	68020	68851	
PTRAPGS	68000	68010	68020	68851	
PTRAPIC	68000	68010	68020	68851	
PTRAPIS	68000	68010	68020	68851	
PTRAPLC	68000	68010	68020	68851	
PTRAPLS	68000	68010	68020	68851	
PTRAPSC	68000	68010	68020	68851	
PTRAPSS	68000	68010	68020	68851	
PTRAPWC	68000	68010	68020	68851	
PTRAPWS	68000	68010	68020	68851	
PVALID	68000	68010	68020	68851	.L
RESET	68000	68010	68020		
ROL	68000	68010	68020		.B .W .L
ROR	68000	68010	68020		.B .W .L
ROXL	68000	68010	68020		.B .W .L
ROXR	68000	68010	68020		.B .W .L
RTD	68010	68020			
RTE	68000	68010	68020		
RTM	68020				
RTR	68000	68010	68020		
RTS	68000	68010	68020		
SADD				TERN	.S .B .W .L .D .X

Instruction	Valid Machine Types				Legal Suffixes
SATAN				TERN	.S .B .W .L .D .X
SBCD			68020		.B
SCC	68000	68010	68020		.B
SCOS				TERN	.S .B .W .L .D .X
SCS	68000	68010	68020		.B
SDIV				TERN	.S .B .W .L .D .X
SECT	68000	68010	68020		
SEQ	68000	68010	68020		.B
SETOX				TERN	.S .B .W .L .D .X
SF	68000	68010	68020		.B
SGE	68000	68010	68020		.B
SGT	68000	68010	68020		.B
SHI	68000	68010	68020		.B
SLE	68000	68010	68020		.B
SLOGN				TERN	.S .B .W .L .D .X
SLS	68000	68010	68020		.B
SLT	68000	68010	68020		.B
SMI	68000	68010	68020		.B
SMOD				TERN	.S .B .W .L .D .X
SMUL				TERN	.S .B .W .L .D .X
SNE	68000	68010	68020		.B
SPL	68000	68010	68020		.B
SSIN				TERN	.S .B .W .L .D .X
SSQRT				TERN	.S .B .W .L .D .X
SSUB	68000	68010	68020		.S .B .W .L .D .X
ST	68000	68010	68020		.B
STAN				TERN	.S .B .W .L .D .X
STOP	68000	68010	68020		

Instruction	Valid Machine Types			Legal Suffixes
SUB	68000	68010	68020	.B .W .L
SUBA	68000	68010	68020	.W .L
SUBI	68000	68010	68020	.B .W .L
SUBQ	68000	68010	68020	.B .W .L
SUBX	68000	68010	68020	.B .W .L
SVC	68000	68010	68020	.B
SVS	68000	68010	68020	.B
SWAP	68000	68010	68020	.W
TAS	68000	68010	68020	.B
TRAP	68000	68010	68020	
TRAPCC	68020			.W .L
TRAPCS	68020			.W .L
TRAPEQ	68020			.W .L
TRAPF	68020			.W .L
TRAPGE	68020			.W .L
TRAPGT	68020			.W .L
TRAPHI	68020			.W .L
TRAPLE	68020			.W .L
TRAPLS	68020			.W .L
TRAPLT	68020			.W .L
TRAPMI	68020			.W .L
TRAPNE	68020			.W .L
TRAPPL	68020			.W .L
TRAPT	68020			.W .L
TRAPV	68000	68010	68020	
TRAPVC	68020			.W .L
TRAPVS	68020			.W .L
TST	68000	68010	68020	.B .W .L

Instruction	Valid Machine Types			Legal Suffixes
UNLK	68000	68010	68020	
UNPK	68020			
USING	68000	68010	68020	

Appendix C

TERN Floating-Point Instruction Set

This appendix provides a list of TERN Floating-Point instructions. The first column lists the instruction by name. The second column lists the format of the instruction. The third column provides an explanation of the instruction. Because the first half of this list of TERN instructions is a *subset* of the 68881 instruction set, you should use this appendix as a reference in conjunction with the 68881 instruction set manual. The second half of the list consists of TERN instructions that *supercede* the 68881 instruction set. Use the TERN instruction whenever the two manuals differ.

Instruction		Format	Meaning
FMOVE	{S|D|B|W|L}	<ea>, FPn	Move data to a floating-point unit.
FMOVE	{S|D|B|W|L}	FPn, <ea>	Move data from a floating-point unit.
FMOVE		FPn, FPm	Move data within a floating-point unit.
FABS	{S|D|B|W|L}	<ea>, FPn	Absolute value.
FADD	{S|D|B|W|L}	<ea>, FPn	Add.
FATAN	{S|D|B|W|L}	<ea>, FPn	Arc tangent.
FBcc		<label>	Branch on condition. Conditions supported are : T,F,EQ,NEQ,GT,GE,LT, LE,GL.
FCMP	{S|D|B|W|L}	<ea>, FPn	Compare.
FCOS	{S|D|B|W|L}	<ea>, FPn	Cosine.
FDIV	{S|D|B|W|L}	<ea>, FPn	Divide.
FETOX	{S|D|B|W|L}	<ea>, FPn	e to the x.
FGETEXP	{S|D|B|W|L}	<ea>, FPn	Get exponent.
FGETMAN	{S|D|B|W|L}	<ea>, FPn	Get mantissa.
FINT	{S|D|B|W|L}	<ea>, FPn	Integer part.

Instruction		Format	Meaning
FLOGN	{S|D|B|W|L}	<ea>, FPn	Log to the base e. Note that this instruction works incorrectly on '881.
FMOD	{S|D|B|W|L}	<ea>, FPn	Module remainder.
FMOVE	{L}	<ea>, FPCONTROL	Move to control register.
FMOVE	{S|D|B|W|L}	<ea>, FPn	Move - Implicit Convert to Internal Floating Point Format (IFP)
FMOVE	{L}	<ea>, FPSTATUS	Move to status regiser.
FMOVE	{L}	FPCONTROL, <ea>	Move from control register.
FMOVE	{S|D|B|W|L}	FPn, <ea>	Move - Implicit Convert of Internal Floating Point Format to specified external format
FMOVE		FPn, FPm	Move - No conversion.
FMOVE	{L}	FPSTATUS, <ea>	Move from status register.
FMUL	{S|D|B|W|L}	<ea>, FPn	Multiply.
FNEG	{S|D|B|W|L}	<ea>, FPn	Negate.
FSCALE	{S|D|B|W|L}	<ea>, FPn	Scale exponent by integer.
FScc		<ea>	Set according to condition. Conditions supported are : T,F,EQ,NEQ,GT,GE,LT,LE,GL.
FSGLDIV	{S|D|B|W|L}	<ea>, FPn	Single Precision divide.
FSGLMUL	{S|D|B|W|L}	<ea>, FPn	Single Precision multiply.
FSIN	{S|D|B|W|L}	<ea>, FPn	Sine.
FSQRT	{S|D|B|W|L}	<ea>, FPn	Square root.
FSUB	{S|D|B|W|L}	<ea>, FPn	Subtract.
FTAN	{S|D|B|W|L}	<ea>, FPn	Tangent.
FTST	{bwlsd}	<ea>	Test.

The following instructions supercede the 68881 instruction set

SADD	{S|D|B|W|L}	<ea>, FPn	Single Precision add. TERN only.
SATAN	{S|D|B|W|L}	<ea>, FPn	Single Precision arc tangent. TERN only.
SCOS	{S|D|B|W|L}	<ea>, FPn	Single Precision cosine. TERN only.
SDIV	{S|D|B|W|L}	<ea>, FPn	Single Precision divide. TERN only.
SETOX	{S|D|B|W|L}	<ea>, FPn	Single Precision e to the x. TERN only.
SSIN	{S|D|B|W|L}	<ea>, FPn	Single Precison sine. TERN only.
SLOGN	{S|D|B|W|L}	<ea>, FPn	Single Precision log to the base e. TERN only.
SMOD	{S|D|B|W|L}	<ea>, FPn	Single Precision modulo remainder. TERN only.
SMUL	{S|D|B|W|L}	<ea>, FPn	Single Precision multiply. TERN only.

Instruction		Format	Meaning
SSQRT	{S|D|B|W|L}	<ea>, FPn	Single Precision square root. TERN only.
SSUB	{S|D|B|W|L}	<ea>, FPn	Single Precision subtract. TERN only.
STAN	{S|D|B|W|L}	<ea>, FPn	Single Precison tangent. TERN only.

Appendix D

Using the Low-Level Debuggers

This chapter describes how to use DB, the low-level debugger, and MDB, the machine level mode of DEBUG, the source level debugger. We present the following topics:

- DB invocation
- DB commands
- Machine Level Debugger invocation under DEBUG
- MDB commands
- Additional debug commands
- Hints for debugging assembly routines

The DB Debugger is the DOMAIN debug utility for DOMAIN assembly language programming. Like the language-level debugger, you can invoke DB from within a Shell. The first part of this chapter describes how to invoke DB and how to use the DB commands.

DEBUG, the DOMAIN source level debugger, has a machine-level debugging sub-mode that we do not document in the *DOMAIN Language Level Debugger Reference* manual. This machine-level debugger is called MDB, which is discussed in second half of this chapter.

D.1 DB Invocation

To invoke DB from a Shell, type the DB command on the command line, in this format:

```
$  DB  pathname
```

where *pathname* is the pathname of your DOMAIN assembly language program. For example:

```
$ db string.bin

Section Map:
 #  Location   Size    Name
  1 002A447C 0000003A PROCEDURE$
  2 002A44B8 00000010 DATA$
Start Address = 002A44B8

2A44B8: 4EF9
```

DB displays a section map by default. The section map provides information such as: the number of sections within a program (there are generally two, by default); the location address of the sections; the size of the sections; and, the name of the section. DB also displays the address and the hexadecimal representation of the first instruction. For example, `2A44B8: 4EF9` corresponds to the location address of START in our program and to the hexadecimal representation of the instruction in the listing file, as shown below.

```
                          (0023)              DATA
    000000> 4EF900000000  (0024) START        jmp      code_start
```

Using DB, you can then examine and/or set memory and registers and execute the program by single stepping or using breakpoints. We describe the basic set of DB commands in the next section. Note that there are a number of additional DB commands that are are highly specialized and oriented toward operating system debugging and crash analysis.

D.2 DB Commands

Once you invoke DB, you can use the commands in the list below. To use a command, type the command after the prompt (!).

Command	Function
A *[<size_spec>] <location>[base_spec]*	Accesses *<location>* and prints the address and contents according to *<size_spec>* and *<base_spec>*.
B *[<location>]*	Sets/clears the breakpoint at the specified location. Breakpoint is not inserted until the G command is given. Previous instruction is reinstalled on the breakpoint entry or vector entry. Only one breakpoint can be defined at the same time.
C *<start> <end> <target>*	Copies memory defined by bounds *<start>* to *<end>* onto memory starting at *<target>* through *<target>+(<end>−<start>)*.
CA *<start>*	Calls the subroutine that starts at *<start>*. All registers saved from the last entry except A0 are restored immediately prior to the call.

Command	Function
F *<start> <end> [<word>]*	Fills memory defined by the bounds *<start>* to *<end>* with a word value *<word>*.
FA	Displays last fault address.
FC	Displays last fault code.
FP{*.s*|*.d*}*[<#>]*	Shows any or all floating-point registers.
FPC *[<value>]*	Sets/Shows the floating-point control register.
FPS *[<value>]*	Sets/Shows the floating-point status register.
G *[<location>]*	Jumps to *<location>* after inserting a breakpoint (if any), restoring all registers and SR.
HELP	Prints DB help file.
PC	Displays current program counter.
Q	Exits from DB.
S *[<size_spec>] <start> <end> <value> [<mask>][<base_spec>]*	Searches memory defined by the bounds *<start>* to *<end>* for *<value>* through an optional *<mask>*. If *<mask>* is not specified, it defaults to $FFFFFFFF. The *<size_spec>* controls the item-size to be searched: byte, word, or long.
SS	Performs single step.
TB	Performs traceback of current stack.

D.2.1 DB Command Formats

```
<size_spec> ::= :I|:B|:W|:L

<location> ::= <address>|Dn|An|CCR|SR
               (An)|<num>(An)|<address>(index_spec)|
               <num>(An,<index_spec>)

<address> ::= <num>|*|<address>+<num>|<address>-num>

<num> ::= <simple_number|$<simple_number>|
          <base>$<simple_number>|-<num>|<quoted_string>

<base> ::= <simple_number>

<quoted_string> ::= '<letter> . . . <letter>' [up to four]

<index_spec> ::= An.W|Dn.W|An.L|Dn.L

<base_spec> ::= :O|:D|:H|:A
```

D.2.2 DB Command Semantics

Locations are evaluated to a memory location or to a saved register, [e.g., Dn, An] or to a location computed from a saved register *[num(An)]*. The meaning of the *size_specs* are

```
:I = instr-size items, output in mnemonic format.

:B = byte-size items, output in numeric format.

:W = word-size items, output in numeric format.

:L = longword-size items, output in numeric format.
```

The meaning of the *base_specs* are

```
:O = numbers and immediate constants printed in octal.

:D = numbers and immediate constants printed in decimal.

:H = numbers and immediate constants printed in hexadecimal.

:A = numbers and immediate constants printed in ASCII.
```

All numeric input defaults to hexadecimal. $num implies hexadecimal. *<base>$num* implies that base is *<base>* [8$777 is octal, 2$1001 is binary]. *<base_spec>* and *<size_spec>* may be specified anywhere in the command line as well as anywhere in A command input (except in quoted strings). All addresses and offsets are printed in hexadecimal regardless of *<base_spec>*.

D.3 Machine Level Debugger Invocation under DEBUG

To invoke the Machine Level Debugger (MDB), type the mdb command at the DEBUG prompt as shown:

```
> mdb
      PC : 000080B8       -- current PC address is displayed automatically
mdb>                      -- MDB prompt
```

MDB has its own set of commands, which we describe below. If MDB doesn't recognize a command, it passes it back to DEBUG for execution. Thus, most DEBUG commands can be entered from the MDB prompt. However, a GO command's execution is deferred until after you exit MDB. This can be confusing; thus, you should generally avoid using GO from within MDB.

MDB commands are case-insensitive and, like DEBUG commands, can be abbreviated to their first letter. Note that MDB commands are similar to a subset of DB commands. However, they are not identical. The following section illustrates their differences.

D.4 MDB Commands

The following is a list of MDB commands.

ACCESS -- Examine or change memory or register value

Format: ACCESS <address-expression>[:<format>][:<size>]

This is the basic command for examining and changing memory or registers. ACCESS displays 1 byte, word, or long word, and then waits for input on the same line. At this point , you can use one of the following options:

<RETURN> Access next word.

/<RETURN> Terminate access, return to MDB prompt.

<value><RETURN> Replace the value with the new one and access next word.

<value>/<RETURN> Replace the value with the new one and terminate.

The <address-expression> can be one of the following:

absolute address A hexadecimal number with 8 or fewer digits. The first digit must be 0–9; append a leading zero if necessary.

register A0–A7, D0–D7, DB(=A5), SB(=A6), SP(=A7), PC.

(register) Indirect register. Register must be A0–A7.

offset(register) Indirect register with hexadecimal offset

By default, ACCESS displays 1 word (2 bytes) of data in hexadecimal format. You can change this format with the **format** and **size modifiers**. The **format options** are

:H hexadecimal (default)
:D decimal
:O octal
:A ascii
:I instruction

The **size options** are

:B byte
:W word (default)
:L long

Although hexadecimal and word are the initial defaults, once you specify a different format and/or size, subsequent ACCESS commands in the same MDB session use those values.

The instruction format option decodes the value as an assembler-like machine instruction. The size option is ignored in this case.

If you enter a new value for a word, you must enter the value in the same format as it is displayed. You cannot enter symbolic machine instructions; MDB does not include an assembler.

ACCESS Examples

```
mdb> a 100CC                          Examine word of memory
000100CC : 6465 /
mdb> a 100CC:l                        Long word at same address
000100CC : 64656275 /
mdb> a 100CC:a                        In ASCII format
000100CC :  debu
mdb> a d0:l:h                         Examine a register
      D0 : 03380003 /
mdb> a (sp)                           Look at top of stack
03386B22 : 81020338                   <RETURN> to access next word
03386B26 : 00030000                   Again
03386B2A : 00000338 /
mdb> a pc:l                           Examine PC
      PC : 000080FC /
mdb> a (pc):i                         Examine next instruction
E - <addr> specifier expected         Unfortunately (PC) is illegal
mdb> a 80FC:i                         Try again with address
000080FC : TST.w       (8.w,a6)       RETURN> to access next instruction
00008100 : BLE.b       811A           Again
00008102 : SUBQ.l      #2,a7 /
mdb> va x                             Get address of variable from DEBUG
FOO\BAR\x = 16-bit integer, local.    VA = 3386B12
mdb> a 3386B12:w:d                    Examine it in decimal
03386B12 : 2            -1 /          Change its value
mdb> a 3386B12                        Check the change
03386B12 : -1           /             Note :d remembered
```

SS -- Single Step Instruction

Format: SS [<address-expression>]

This command single-steps one machine instruction, then decodes and displays the next instruction. If you specify an address, the PC is set to the address before stepping.

Examples

```
mdb> a pc:l
      PC : 000080FC /
mdb> s                                -- Execute instruction at 80FC
00008100 : BLE.b       811A           -- This is *next* instruction
mdb> s                                -- Execute it
00008102 : SUBQ.l      #2,a7
mdb> s 811A                           -- Force branch to 811a
00008104 : MOVE.w      (8.w,a6),d0
```

WALK -- Step Multiple Instructions

Format: WALK <n>

This is equivalent to n successive SS commands.

Example

```
mdb> w 3
0000807E : MOVE.w      d0,(-A.w,a6)
00008082 : MOVE.w      #7,-(a7)
00008086 : PEA.l       (24.w,PC)
```

HELP -- Display On-Line Help Information

```
Format:  HELP
```

Displays a summary of MDB commands.

QUIT -- Terminate MDB

```
Format:  QUIT
```

Terminates MDB and returns to DEBUG command level.

D.5 Additional Debugging Commands

Two additional low-level debugging commands are available in DEBUG:

REGS displays all machine registers (except floating-point), plus some additional machine status information.

FPREGS displays all floating-point registers on workstations that have FP registers. Note that MDB cannot access floating-point registers.

D.6 Hints for Debugging Assembler Routines

DEBUG normally steps over assembler routines when it encounters them. The simplest technique for debugging an assembler language procedure is to stop the calling program just before the call, enter MDB, and single-step into the procedure.

You can set breakpoints in assembler code using the -VA option of the DEBUG BREAKPOINT command. You will probably have to use MDB to find the address at which to break, since DEBUG does not know about assembler routines by name. (The -SMAP DEBUG startup option may also be useful.) When the program stops in assembler code DEBUG will not be able to establish a valid environment and many commands will not be available. However MDB can always be invoked.

The DEBUG VA command is useful for determining the addresses of variables, etc., to use with MDB. For more information about DEBUG, refer to the *DOMAIN Language Level Debugger Reference*.

Appendix E

Pre-SR9.5 Calling Conventions

At SR9.5, some of the routine and calling conventions were changed to reduce procedure call overhead and to permit compilers to take advantage of optimization techniques that require register saving across calls.

This appendix describes the DOMAIN calling conventions prior to Software Release 9.5 (SR9.5). Use this appendix if you need information about pre-SR9.5 modules.

E.1 The Stack

The **stack**, an area of temporary data storage, grows from high memory addresses to low memory addresses. The two stack-related registers are **A6**, the stack frame register, and **A7**, the stack pointer. Address register A6, or Stack Base (SB) points to the current stack frame. Address register A7, or Stack Pointer (SP) points to the top of the stack. The area of memory between A6 and A7 constitutes the **automatic storage**, or local variables of the current routine.

E.1.1 Stack Format

As an example, Figure E–1 illustrates how the stack format looks if routine *Finder* calls routine *Seeker*.

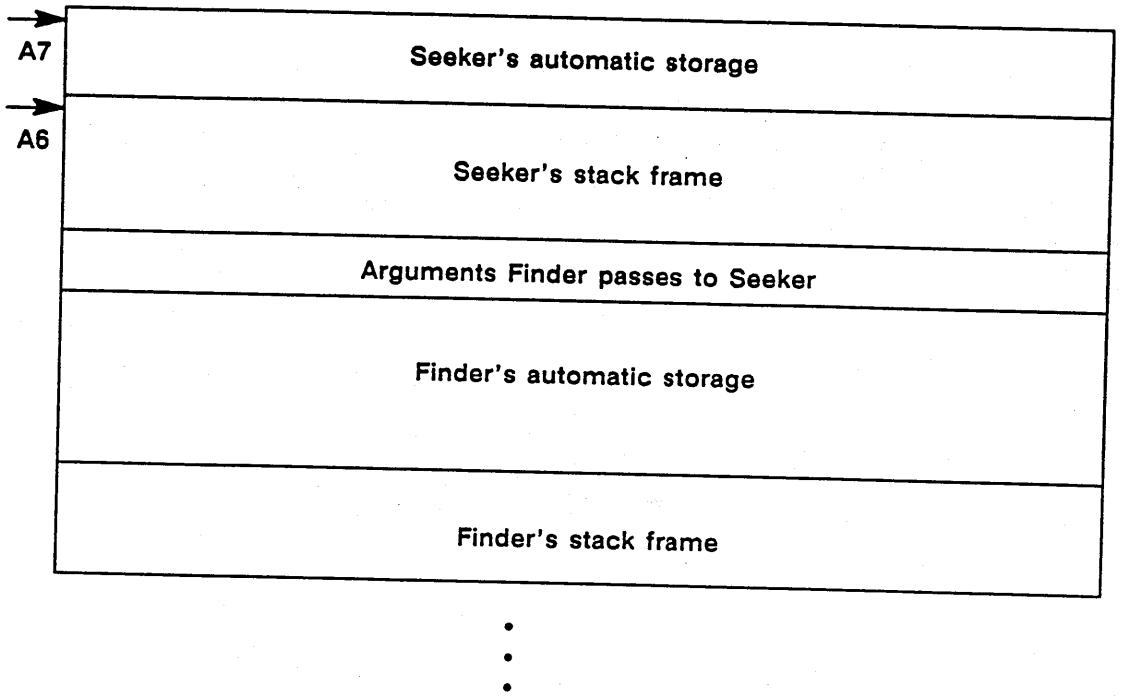


Figure E–1. Stack Format

E.1.2 Stack Frame Format

Figure E–2 illustrates the component fields of the stack frame. The A6 register points to the stack frame. Following the illustration, we discuss each field in detail.

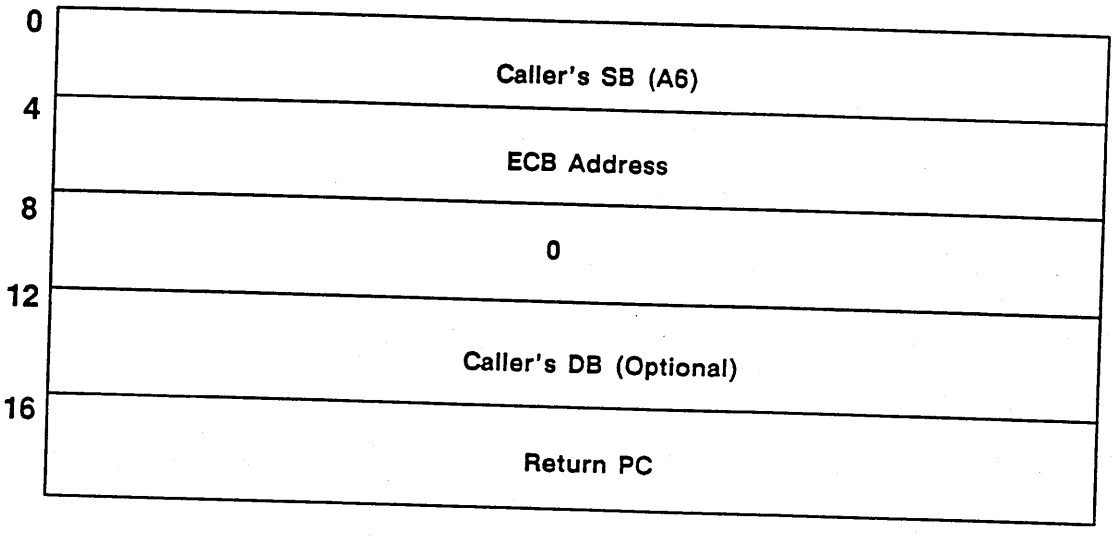


Figure E-2. Stack Frame Format

Caller's SB Field

The 4-byte Caller's SB field is a pointer to the previous stack frame. Pushing the caller's SB onto the stack preserves the SB across the call and provides a way for the data to thread back through the stack.

ECB Address Field

The 4-byte ECB (Entry Control Block) address field in *Finder's* stack frame (refer to our example on the previous page) is the address of *Finder's* ECB. Language-level debug commands such as **TB** and **DEBUG** use this field to print a **stack trace**, which is the routine name and line number. The stack frame has the address of the ECB, which in turn contains the address of the debug tables.

0 Value Field

This 4-byte field is set to zero by the prologue code. Some DOMAIN software checks this field to ensure it is a valid stack frame.

Caller's DB (Data Base) Field -- (Optional)

This 4-byte optional field contains the caller's DB or A5 register. To determine if the field is present, look at the flag word of the routine's ECB. If the *B* bit is reset to zero, the field is present. Refer to the "ECBs" section within this appendix for more information.

NOTE: The fields described above are set up by the called routine's **prologue code**. We discuss both the prologue and the epilogue code in the next section.

Return PC Field

A **JSR** instruction pushes the Return PC onto the stack and transfers control to the routine. An **RTS** instruction returns control when the called routine finishes.

E.1.3 Prologue Code and Epilogue Code

This section describes the prologue code and epilogue code.

Prologue Code

Prologue code consists of the instructions that DOMAIN compilers generate at the beginning of a routine to set up the stack frame. The standard prologue code is shown in Figure E–3.

```
PROC_STRT       MOVE.L    DB,-(SP)      Save caller's A5 register (DB).
                CLR.L     -(SP)         Push 0 value field.
                MOVE.L    A0,-(SP)      A0 contains ECB address.
                MOVE.L    6(A0),DB      Load caller's A5.
                LINK      SB,#-A_SIZE   Save caller's SB, set up
                                        automatic storage area.
```

Figure E–3. Standard Prologue Code

NOTE: The **MOVE** instruction that pushes the caller's DB is not present if the *B* bit in the ECB flag word is 1.

Note that `#-A_SIZE` is the size of the automatic storage needed by the routine. The minus sign indicates that the stack grows downward in memory.

Epilogue Code

Epilogue code consists of instructions that the DOMAIN compilers generate to return from a routine. The standard code that is used to return from a called routine is shown (with comments) in Figure E–4.

```
                UNLK      SB            Restore caller's SB.
                ADD.W     #8,SP         Pop ECB address and 0 field.
                MOVE.L    (SP)+,DB      Restore caller's DB.
                RTS                     Return.
```

Figure E–4. Standard Epilogue Code

NOTE: The **MOVE** instruction that restores the caller's DB is not present if the *B* bit in the ECB flag word is 1.

Note that registers A0 and D0 can contain values on function return.

E.1.4 Calling a DOMAIN Assembly Language Routine

The following example is based on the information given in the previous section. Here we see how a high-level program calls an DOMAIN assembly language routine.

FORTRAN PROGRAM

```
C common.ftn
        common/abc/c
        integer c
        c = 4
        write(*,*) c
        call common
        write(*,*) c
        end
```

DOMAIN Assembly Language Routine

```
* common.asm
* Example of accessing FORTRAN named COMMON in assembly language
* common.asm uses DFSECT, SECT, and DS pseudo-ops to model
*        COMMON/ABC/C is INTEGER
* common.asm sets C to 5
*
        module  common
        entry   common
pro     equ     *
        move.l  db,-(sp)        *  Prologue Code
        clr.l   -(sp)                    .
        move.l  a0,-(sp)                 .
        move.l 6(a0),db                  .
        link    sb,#0                    .
        move.l ac_c,a1
        move.l #5,(a1)
        unlk    sb              *  Epilogue Code
        add.w   #8,sp                    .
        move.l  (sp)+,db                 .
        rts                              .
abc     dfsect  overlay
        sect    abc
c       ds.l    1                  reserve 4 bytes for integer
        data
common  jmp.l   pro             *  Entry Control Block
        ac      common                   .
        dc.w    0                        .
ac_c    ac      c
        end
```

The FORTRAN program prints the value of C, which is equal to 4. Then, the program calls the DOMAIN assembler routine and prints the value of DOMAIN Assembler's C, which is 5. The ECB is described in the "ECBs" section.

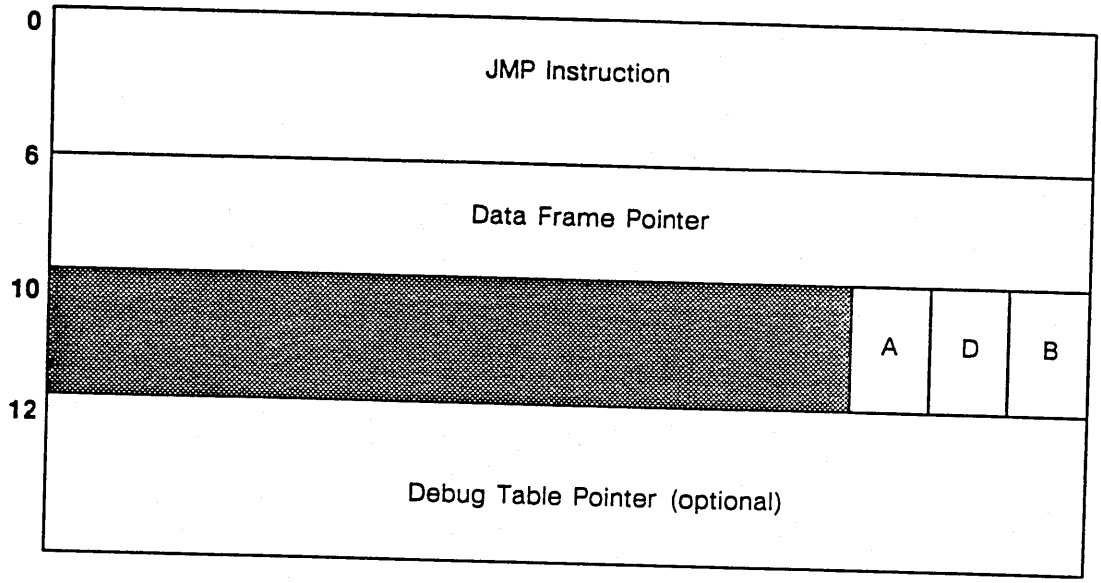
E.1.5 Notes on Register Conventions

From our discussion at the beginning of the appendix, we know that A7 points to the top of the stack and A6 points to the current stack frame. However, A5 or Data Base (DB) is also very important in the calling sequence. A5 is preserved (saved and restored) across function and procedure calls; conversely, D0–D7 and A0–A4 *may not* be preserved. When DOMAIN compilers generate code, they use A5 to point to the frame containing static data and linkages to external references. DOMAIN–generated code expects A0 to contain the ECB address on entry to a function or procedure. A0 and D0 are used to return function results. A5, A6, and A7 are preserved across a function or procedure call.

E.2 ECBs (Entry Control Blocks)

DOMAIN compilers generate an **ECB**, or **Entry Control Block**, for every function and procedure definition. The ECB address is the value of the object module global that corresponds to an external function or procedure. This convention is true for both standard and C calling conventions.

Because ECBs require relocation at load time, they are located in the read/write sections. Figure E–5 illustrates the ECB format. An assembly language fragment containing an ECB follows the illustration.




```
                    . . .
                    PROC
                    ENTRY.P   SEEKER
PROC_STRT           EQU       *
*           prologue and epilogue code go here.
                    . . .
                    DATA
DATA_STRT           EQU       *
                    . . .
*           Entry Control Block starts here.
SEEKER              EQU       *                 Entry is ECB.
                    JMP.L     PROC_STRT         Prologue Code.
                    AC        DATA_STRT         Data Frame Pointer.
                    DC.W      0
                    . . .
```

Figure E-5. ECB Format and Example

An ECB is an example of code that must reside in a read/write section (because it requires relocation). The ECB can be either 12 or 16 bytes in length, depending on whether or not the debug table pointer is present. We discuss each field below.

JMP Instruction Field

This 6-byte field, which contains the **JMP** instruction, references the prologue code in the procedure frame associated with the routine, as shown in Figure E-5 above. The **JMP** instruction uses the long absolute addressing mode. Therefore, the first two bytes of the instruction are 4EF9 (hex) and the remaining four bytes are the absolute address of the prologue code.

Data Frame Pointer Field

The 4-byte data frame pointer field contains the address of the data frame. The data frame contains static read/write data, linkages to external data and routines, and the ECB. The prologue code loads the A5 (DB) register with the data frame pointer (refer to the "Prologue and Epilogue Code" section within this appendix). The routine uses the DB register to reference the static data and linkage information, such as address constants, in the data frame.

B Bit

The rightmost bit, or *B* bit, indicates whether the prologue code for this routine saves the caller's A5 register in the stack frame. A 0 in this position indicates that the caller's A5 register is saved in the stack frame. If a 1 is present in this position, the caller's A5 register is not saved.

D Bit

The *D* bit indicates whether the next field in the ECB, the debug table pointer field, is present. A 1 in this position indicates that the debug table pointer field is present; a 0 indicates that the field is absent.

A Bit

The *A* bit indicates that this routine is a part of DOMAIN-supplied software. The language-level debugger uses this to keep DEBUG from stepping into DOMAIN-supplied software.

Debug Table Pointer Field (Optional)

This 4-byte optional field contains the address of the start of the debug tables associated with this routine. DOMAIN compilers generate the debug tables in a read-only section called *DEBUG$* in the object module. The *D* bit, described above, indicates whether this field is present.

E.3 Passing Parameters

The conventions for passing parameters are the same for SR9.5 and pre-SR9.5 code. Refer to Chapter 6 for details.

Appendix F

The Object Module

The DOMAIN compilers and the DOMAIN assembler translate your source programs into **object module format.** When you use the binder utility to bind one or more object modules, the binder produces a single object module. Both your input modules and the binder's output module use the same object module format.

This chapter divides the discussion of object modules into two parts: overview and application. Basically, the overview provides a theory of operation, in which we discuss the role the loader and binder play, and how to display and interpret an object module listing. The second part of the chapter uses a sample object module to illustrate the object module format we discuss in detail. Our example in this chapter is based on a FORTRAN source file and an DOMAIN assembly language source file. The overview topics are:

- What the binder does
- What the loader does
- Producing an object module listing
- Interpreting the object module listing

After the overview, we explore the object module elements in detail and provide an example object module listing for reference.

F.1 What the Binder Does

Some of the functions that the binder performs in the output object module are as follows:

- Binds multiple object modules into a single object module.
- Resolves external or global references and definitions.
- Sets the nonreplacable attribute in the **section attributes field.**

- Modifies the **install and look at installed attributes**, if necessary, in the section attributes field.
- Satisfies the **alignment attribute** in the section attributes field by ensuring the read-only sections' position in the output object module.
- Uses the **old global field** to select a global definition when multiple definitions of the same global name occur.
- Copies **module information records** from input object modules and adds a **maker module information record.**
- Combines **static resource information** from input object modules.
- Prints the information in the module information maker records if you use the binder option **-MAKER.**
- Combines sections with the same name.

We discuss the boldfaced terms and binder implications in greater detail where applicable in this chapter.

F.2 What the Loader Does

The **loader** loads the object module into memory by performing the following functions:

- Maps read-only sections to memory.
- Allocates space for read/write sections.
- Resolves references to installed libraries using KGT (Known Global Table). When you install libraries in the address space, their entry points are put into KGT. We discuss this table in greater detail later at the end of this chapter.
- Initializes data and relocates address in read/write sections.

F.3 Producing an Object Module Listing

Once the assembler, compiler, or binder translates your program into an object module, you can list information about the object module using the OBJDMP AEGIS Shell command. The format of the command is:

```
$ OBJDMP input filename [-option(s)]  [outputfile]
```

where the object module (filename.bin) is the input file and a text file is the output file. The output filename is optional (except where noted); use an output filename if you want to keep a copy of the object module listing. If you do not use an output filename, OBJDMP dumps the listing to *stdout* (standard output).

The OBJDMP command options are listed below. If you do not include any options, the OBJDMP displays *all* information.

Option	Meaning
-L[IST]	Write information to the indicated file. You must specify an output filename.
-H[EADERS]	Display header information.
-SEC[TIONS]	Display the section table.
-G[LOBALS]	Display the globals table.
-SRI	Display the SRI records.
-MAKER	Display the maker records.
-IMP[URE]	Display the impure data.
-DEBUG	Display debug information.

F.4 Interpreting the Object Module Listing

The following examples illustrate a FORTRAN source program and the object module listing. Throughout the section, "Object Module Elements" section that follows, we refer to the appropriate part of the object module listing example to illustrate our points.

Example 1a: FORTRAN Source Program

```
subroutine try
common /block/iarray(20)
data iarray/20*1000/
call again
end
```

Example 1b: Object Module Listing

```
Apollo Object Module Dumper -- 2.0
*** Object Module Header ***
Identification = Program_Module
Format_Version =     3
Mapped_Size = 00000054

*** Global Information Header ***
Start_Address =   0 00000000
Name = TRY
Creation_Time = 28FE9802
Time Stamp: 1985/09/17 14:09:08 EDT (Tue)
N_Sections          =     4
N_SRI               =     0
N_Globals       =         2
N_Shared_Libraries =      0


*** Section Index Table ***
 ID  NAME                                LOCATION  SIZE
  1 PROCEDURE$                           00000020  0000001C R/O Concat Instr
Long-aligned
  2 DATA$                                00000000  00000014 Concat Data
Long-aligned
  3 DEBUG$                               0000003C  00000018 R/O Concat Data
Long-aligned
  4 BLOCK                                00000000  00000050 Ovly Data Long-aligned
*** Globals ***
   5 (  0 00000000) AGAIN
   6 (  2 00000004) TRY                                  Marked
*** Static Resource Information ***
*** Impure Data ***
   Text_Rec:
      Id:  2 Text_Mem_Addr: 00000000 Byte_Count:   20
       0000: 00000000 4EF90000 00000000 00140002
       0010: 00000000
```

```
Reloc_Rec:    4
        OFFSET    RELOC_BASE_ID
     1    0000       5
     2    0006       1
     3    000A       2
     4    0010       3

Text_Rec:
    Id:  4 Text_Mem_Addr: 00000000 Byte_Count:     4
     0000: 000003E8
 Repeat_Rec:         20
 End_Rec
```

F.5 Object Module Elements

The object module is composed of the following standard elements:

- Object module header
- Read-only sections
- Global information header
- Section index table
- Global table
- Read/Write section templates

We discuss each standard element in detail in the sections that follow. Optional elements of the object module (*Module Information records, Static Resource Information records,* and *Debug Tables*) are discussed later in the chapter. First, we present an illustration of the object module format.

Within each section, we illustrate the format of the particular element and provide an example from the actual object module listing shown in Example 1b above. The diagrams presented in this section contain numbers to the left of the diagram. These numbers are byte displacements (decimal) of the fields from the beginning of the element.

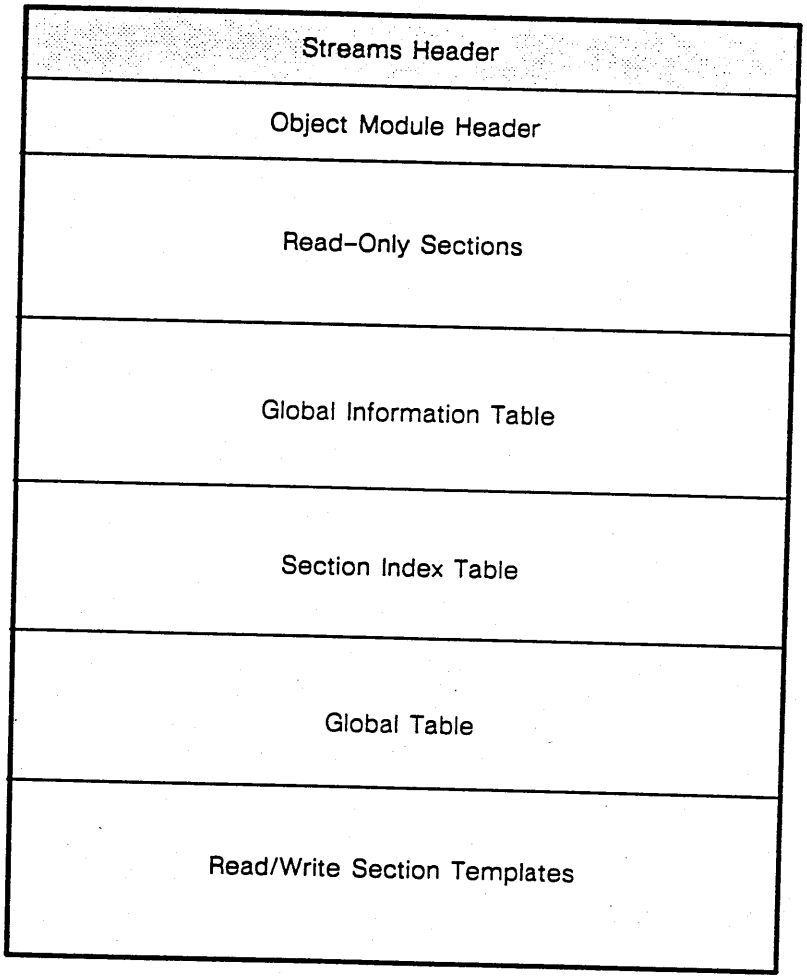


Figure F–1. Object Module Elements Format

NOTE: Object module data is preceded by a 32–byte streams header (shaded area in illustration). This header is transparent if you read the file through a read operation. However, if you map the file, the streams header is included.

F.5.1 Object Module Header

The **object module header** is the first element of an object module. The main purpose of the 32-byte header element is to provide pointers to other elements in the object module. These 4-byte pointers, which are 32-bit integers that give the position in the file relative to the start of the object module, are called **file pointers.** File pointers are offsets from the beginning of the object module header to the current position in the file. The fields in the object module header are shown below in Figure F-2.

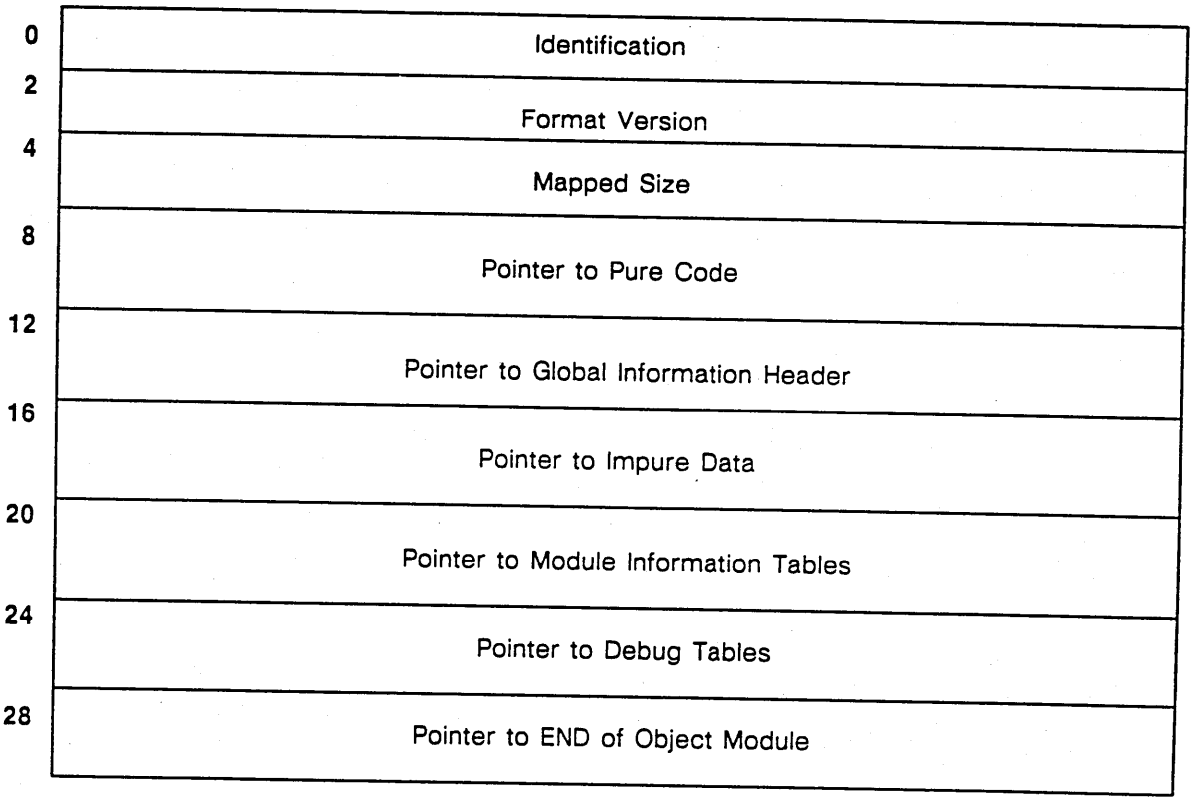


Figure F-2. Object Module Header Fields

We discuss each field below. Use the following example, from the object module listing, for reference as you read about the fields.

```
** Object Module Header ***
Identification = Program_Module
Format_Version =      3
Mapped_Size = 00000054
```

Identification Field

The identification field is a 2-byte field that contains a value identifying the object module as either a program module, a library module, or an object module modified by the debugger. Table F-1 lists the identification value and the object module type.

Table F-1. Identification Field Values

Identification Field Value	Object Module Type
1	**Program Module.** All object modules created by the compilers and binder. This is an executable object module.
2	**Library Module.** Library modules are output by lbr and input to the binder.
4	**Object Module Modified by Debugger.** Occurs if you set breakpoints in a program running under DEBUG with the -NC option. During normal termination, DEBUG removes any breakpoints and sets the ID field value back to 1. Abnormal debug terminations can cause the value to remain at 4.

NOTE: The loader will not execute object modules that contain identification field values other than 1.

In our example above, the subroutine is identified as a `Program_Module`.

Format Version Field

The 2-byte format version field contains a value of 3. For example, `Format_Version = 3`. The format version number identifies the version; the value 2 indicates pre-SR9.5 and 3 indicates SR9.5. Currently, these are the only valid values. We discuss only SR9.5 in this chapter.

Mapped Size Field

The mapped size field is the length in bytes of the read-only sections plus the object module header (32-bytes). The loader maps the object module from the start of the object module header to the end of the read-only sections. The loader uses the mapped size field value as an argument to the mapping call.

In our example, the mapped size is `00000054` bytes (hex).

Pointer to Pure Code Field

The pointer to pure data, pure code, or non-relocatable code, contains the byte displacement of the read-only sections from the start of the object module. The pointer to pure code should be 32 because the read-only sections follow the 32-byte object module header in the object module format. However, we advise using the pointer in case the header size changes.

Pointer to Global Information Header Field

The pointer to global information header contains the byte displacement of the global information header from the start of the object module. This pointer contains the same value as the mapped size field because the global information header follows the read-only sections in the object module format.

Pointer to Impure Data Field

The pointer to impure data field points to the start of the read/write section templates. These templates, described in greater detail in the "Read/Write Section Templates" section later in the chapter, are a series of records. The read/write templates provide information to the loader about initialization and relocation required in the read/write sections. In our Example 1b: Object Module Listing, **DATA$** and **BLOCK** are read/write sections.

Pointer to Module Information Tables Field

This field contains the pointer to the optional module information tables. If the tables are absent, the field value is 0. Otherwise, the field value is the byte displacement of the module information tables from the start of the object module.

When the tables are present, the module information can occur anywhere after the section index table. DOMAIN compilers and the binder put the module information after the read/write templates at the end of the object module. Refer to the "Module Information Records" section within this chapter for more information.

Pointer to Debug Table Field

This field points to the debug table header. For more information, refer to the "Debug Table" section within this chapter. The debug table is separate from the **DEBUG$** section, which also contains debugging information.

Pointer to END of Object Module

The pointer to END of object module contains the value of the length (in bytes) of the object module.

F.5.2 Read-Only Sections

The read-only sections follow the object module header in the object module format. The read-only sections are one of two types of major sections: *read-only* and *read/write*. We describe the read/write section in more detail later in this chapter.

The read-only element can contain multiple sections. However, because read-only sections cannot be modified during loading or execution, they can only contain instructions and constant data (pure code, nonrelocatable or position independent code). The loader maps read-only sections with read and execute-only rights to ensure that the object module is not modified during loading and execution time.

The advantages to mapping the read-only sections include:

- Protection by ensuring that pure code and data is not modified while the program is executing.
- Faster loading time than read/write sections because the loader does not have to read the templates in the object module and initialize the data as in the read/write sections.
- Better working set performance when more than one process is running the same program. The read-only sections of a program can be shared by multiple processes running the same program.

F.5.3 Global Information Header

The global information header follows the read-only sections in the object module format. You will recall that the object module header (described above) contains a file pointer to this element.

Before introducing the global information header element, we provide an overview of globals. **Globals** are intermodule references. Compilers generate object module globals for every external function and data variable.

An object module global can be either a reference or a definition. As an example, a call to an external procedure is a global **reference**. A **definition** can be illustrated as an external procedure body. One of the binder functions is to resolve global references and definitions.

The 74-byte global information header contains the fields shown in by Figures F-3.

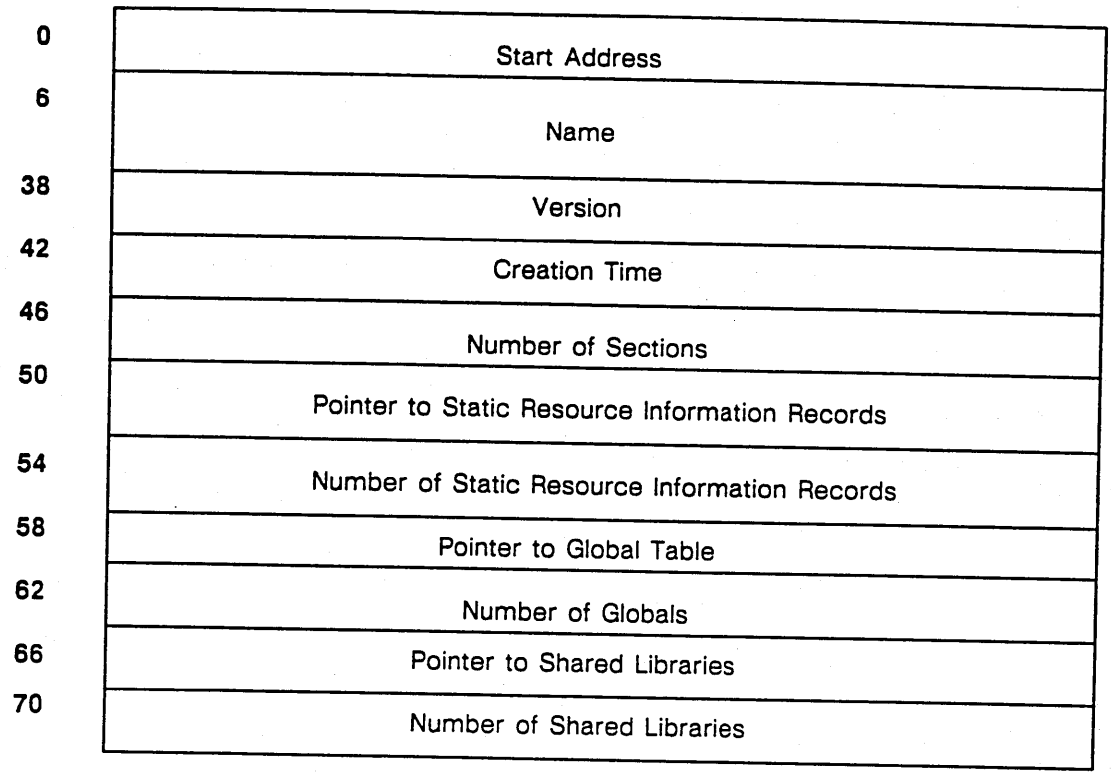


Figure F-3. Global Information Header

We describe each field below. Use the following example for reference.

```
*** Global Information Header ***
Start_Address =   0 00000000
Name = TRY
Creation_Time = 28FE9802
Time Stamp: 1985/09/17 14:09:08 EDT (Tue)
N_Sections          =     4
N_SRI               =     0
N_Globals       =         2
N_Shared_Libraries =      0
```

Start Address Field

The **start address field** identifies the location of the first executable instruction, where the loader transfers control to the program after loading the object module. The 6-byte start address is composed of two fields: a 2-byte section ID and a 4-byte offset. The section ID is the section's position in the section table. Refer to the "Section Index Table" section in this chapter for more information. The offset is the displacement from the start of the section. In our example the ID is 0; the offset is 00000000. This is because our example is a subroutine, which means that the object module does not have a start address.

A program can only have one start address. At binding time, the binder issues a warning message: `Attempt to respecify start addr` if more than one object module contains a start address. For example, the FORTRAN compiler creates subroutine object modules without start addresses. If an object module does not have a start address, the section ID and offset fields are set to 0, as shown in the example above.

Name Field

The 32-byte name field identifies the object module. If the name is shorter than 32 characters, the remaining spaces in the field are blank filled. The name of our example program is TRY.

Version Field

The 4-byte version field identifies the compiler version number. DOMAIN compilers currently set this field to 0.

Creation Time Field

The 4-byte creation time field identifies the time at which the object module is created. DOMAIN compilers set this field to the most significant 4-bytes that the routine *time_$clock* returns. OBJDMP converts the creation time to *date/time* format using *CAL_$* routines and the TS command prints out the creation time as a time stamp.

Number of Sections Field

The 4-byte number of sections field contains the number of sections (both read-only and read/write) within the object module.

In our example, `N_Sections = 4`. Note that this number corresponds to the four sections identified in the section index table part of the listing shown in the next section.

Pointer to Static Resource Information (SRI) Records Field

Static resource information (SRI) records are an optional element in the object module. The 4-byte file pointer, which points to optional SRI records, contains the byte displacement of the first static resource information record from the start of the object module. This field contains a 0 if no static resource information records exist.

Number of Static Resource Information (SRI) Records Field

The 4-byte number of static resource information records field contains the actual number of static information records that follow. This field contains a 0 if no static resource information records exist. For example, `N_SRI = 0`. We explain static information records in the "Static Resource Information (SRI) Records" section in this chapter.

Pointer to Global Table Field

The 4-byte file pointer to the global table field contains the byte displacement of the start of the global table from the start of the object module. We explain the global table in more detail later in this chapter.

Number of Globals Field

The 4-byte number of globals field contains the sum of global definitions and references found in the global table. In our example, the two globals are the subroutine name TRY and the called routine name AGAIN.

Pointer to Shared Libraries Field

This 4-byte field is currently not used. Its value is set to 0.

Number of Shared Libraries Field

This 4-byte field is currently not used. Its value is set to 0. For example, `N_Shared_Libraries = 0`.

F.5.4 Section Index Table

The section index table immediately follows the global information header. The **section index table** is an array of entries consisting of one entry for each section.

Sections have properties called **attributes.** These attributes are represented as bits in a 2-byte field. One attribute in the field determines if the section is read-only or read/write. If the bit in this field is 1, then it is a read-only section; otherwise, it is read/write section.

Sections are referred to elsewhere in object modules using 16-bit integer IDs, as illustrated by the start address in the Global Information header and by the reloc_id in relocation records. The ID of a section is its position in the section table. Each section table entry is 42 bytes long, as shown below in Figure F-4.

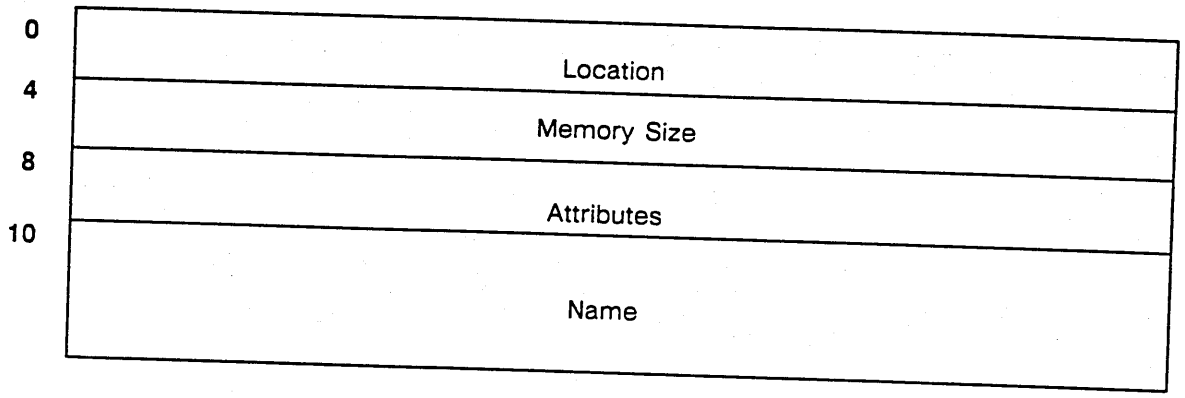


Figure F-4. A Section Table Entry

We describe each field below. Refer to the following example for more information.

```
*** Section Index Table ***
 ID  NAME                          LOCATION  SIZE
  1 PROCEDURE$                     00000020  0000001C R/O Concat Instr
Long-aligned
  2 DATA$                          00000000  00000014 Concat Data
Long-aligned
  3 DEBUG$                         0000003C  00000018 R/O Concat Data
Long-aligned
  4 BLOCK                          00000000  00000050 Ovly Data Long-aligned
```

Location Field

For read-only sections, the location field is a file pointer, or byte displacement of the section, from the start of the object module. For read/write sections, this field is set to 0. In our example, the location of PROCEDURE$ is 00000020 and the location of DEBUG$ is 0003C.

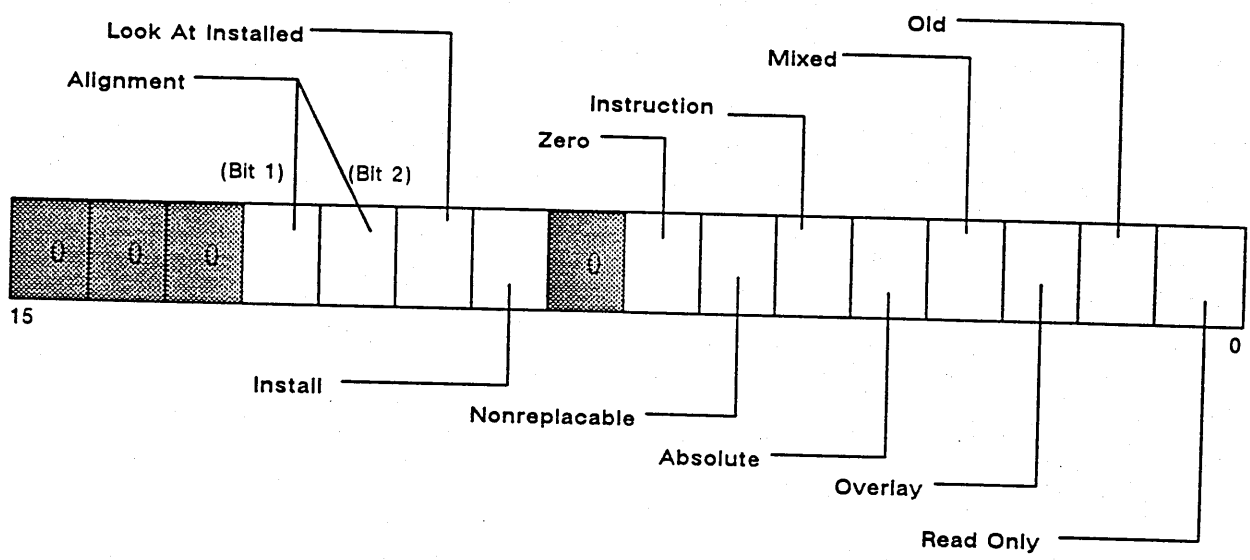
Memory Size Field

The 4-byte memory size field contains the size of the section. Because read-only sections are represented in memory image format, the size of the read-only sections corresponds to its size in the read-only section element in the object module. For read/write sections, this is the amount of memory that the loader allocates. In our example, the size of PROCEDURE$ is 0000001C. The size of DEBUG$ is 00018. The loader uses memory size to determine how much space to allocate during loading for read/write sections.

Attributes Field

The 2-byte attributes field contains several section attributes. Each bit in the field corresponds to a section attribute. Figure F-5 illustrates each bit in the section attributes field. We discuss each attribute in detail.

In the example shown above, the attributes of PROCEDURE$ are written as R/O Concat Instr, which means that the read-only and Instruction attribute bits are set and the overlay bit is reset. Many of these bits can be set/reset with binder options. Refer to the "Section Attributes" section in the *DOMAIN Binder and Librarian Reference* for more information.



NOTE: The shaded bits in the figure are reserved. Therefore, they must be set to 0 in each section attribute field.

Figure F-5. Section Attributes Field

READ-ONLY BIT. This least significant bit identifies the section as read-only if the bit is 1. If the bit is 0, the section is read/write. You can issue limited control of these attributes by using the -READONLYSECTION binder option. Refer to the *DOMAIN Binder and Librarian Reference* for a description of this binder option.

OLD BIT. This bit indicates old sections. If the bit is 1, the section is old. The binder does not copy old sections to the output object module. Also, the loader does not load old sections. Currently, you cannot explicitly mark a section as old.

OVERLAY BIT. The **overlay section** attribute determines how the binder combines sections that have the same name. If the bit is set to 1, then the section is **overlay**. When the binder combines sections with the overlay section attribute, the size of the resultant section is the size of the largest individual input section. You can use an overlay section when several modules define shared data. For example, a FORTRAN *COMMON* block is an overlay section. You have indirect control of these sections through the source code.

If a section is not overlay (the bit is set to 0), the section is **concatenate**. When the binder combines a concatenate section, the size of the resultant section is the sum of the input sections.

MIXED BIT. The binder uses the mixed bit for internal classification. The bit indicates whether a section is declared concatenate in some input object modules or overlay in others. Only the binder can set the mixed bit. This enables the binder to report useful error messages if contradictory declarations occur in a future attempt to bind object modules. In a mixed case, the binder or librarian assumes that the section has the overlay attribute.

ABSOLUTE BIT. The assembler generates an **absolute section** when you define a global with an absolute address, as illustrated by the following example:

```
            . . .
            ENTRY     VAR1
VAR1        EQU       $FE80AB
            . . .
```

Defining a global with an absolute address can be useful when you want to symbolically refer to an address used for memory mapped I/O. Absolute sections have a size field of 0.

The absolute attribute informs the loader to begin the section at a fixed virtual address. In a high-level language, you have no control over this attribute.

INSTRUCTION BIT. The instruction attribute indicates whether a section contains instructions. If a section does not contain instructions (bit is set to 0), it is a data section. Read-only and read/write sections can have an instruction attribute. However, instructions normally are found only in the read-only sections. Currently, only the binder examines the instruction bit. The binder does not set the *install* or *look at installed* attribute in a section that contains the instruction attribute.

NONREPLACABLE BIT. The binder sets this bit if a global is defined as a non-zero offset from the section start and the binder resolves a reference to that global. The binder sets the nonreplacable attribute in the output object module for the section in which the global is defined.

NOTE: Currently, sections cannot be replaced.

ZERO BIT. The zero attribute is used only for read/write sections. If this bit is 1, the loader zeros the section after allocating space for the read/write section and before processing the read/write templates. However, if the bit is 0, the only initialization is specified by the template, which can leave parts of the section indeterminate.

INSTALL BIT. The loader examines this bit when it installs libraries. If the bit is set to 1, the loader enters the section's name, location, and size into the Known Global Table (KGT). Refer to the "Known Global Table" section at the end of this chapter for more information. To share a section between a program and an installed library, the section in the installed library must contain an install bit. You can control the installed attribute with the **-MARKSECTION** and **-NOMARKSECTION** binder options.

LOOK AT INSTALLED BIT. The loader examines this bit in your programs. If the look at install bit is set to 1, the loader checks the Known Global Table (KGT) to see if the section is installed (see above). An install section and a look at installed section match if they have the same name and if the size of the install section is equal to or greater than the look at installed section. To share a section between a program and an installed library, the section in the program must contain the look at installed bit. You can control the look at installed attribute with the **-LOOKSECTION** and **-NOLOOKSECTION** binder options.

ALIGNMENT BITS. Unlike the single-bit attributes described above, this attribute is composed of two bits (refer back to Figure F-5 for the proper position of the bit numbers). Depending on the combinations of setting in the two bits, alignment can vary. Table F-2 lists the combinations and the resultant section alignment boundaries.

Table F–2. Alignment Bits and Section Alignment Boundaries

Bit 1	Bit 2	Section Alignment	Section Alignment Description
0	0	4–byte boundary (Default)	Long–aligned
0	1	8–byte boundary	Quad–aligned
1	0	Reserved for future use	N/A
1	1	Page boundary (1024–bytes)	Page–aligned

The loader aligns read/write sections according to the alignment attributes shown in Table F–2.

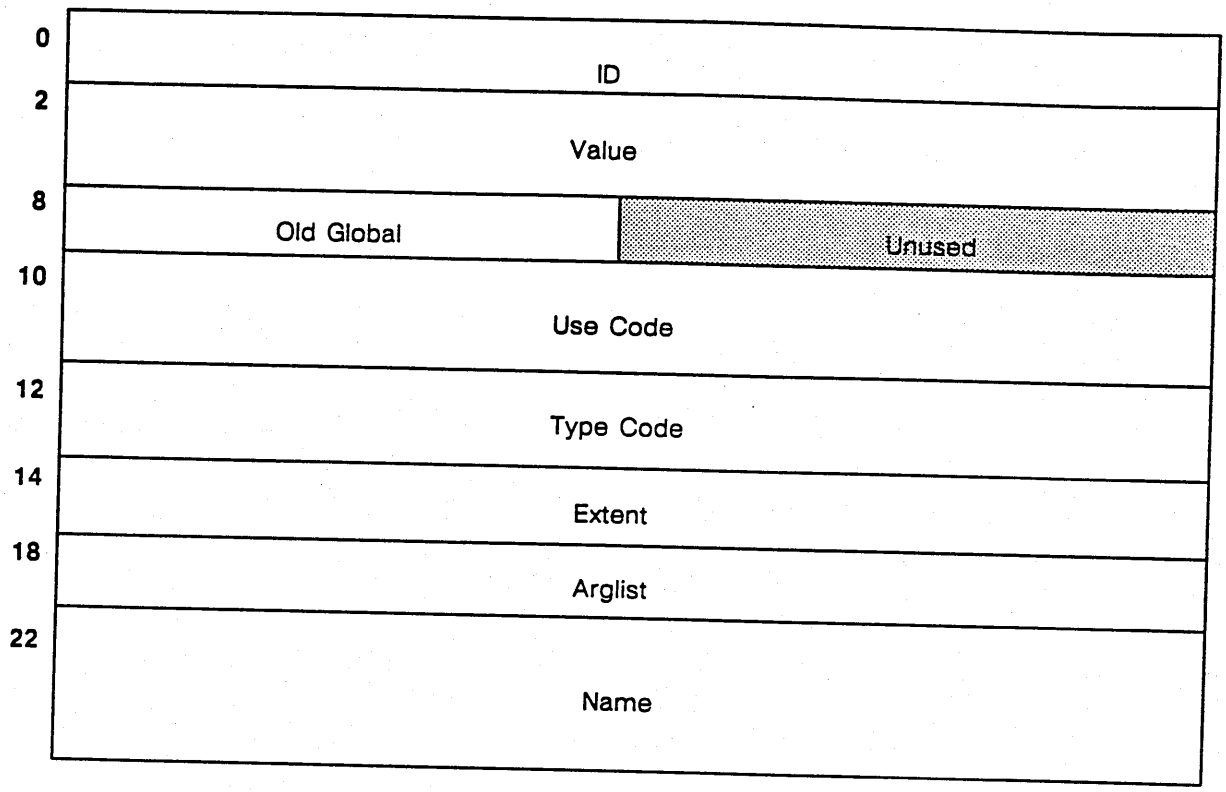
For read–only sections, the binder satisfies the alignment attribute by ensuring the read–only sections' position in the output object module. The pure code region of the object module is always mapped page–aligned. You can control these attributes through the **–ALIGN** binder option. Note that compilers that generate object modules must be concerned about alignment of read–only sections in the object module.

Name Field

The 32–byte name field contains the name of the section. Names shorter than 32 bytes are blank filled. Each section with the object module must have a unique name. Refer to the "Read–Only Section" section for more information regarding names. Our example has three default name sections. The fourth section name, *BLOCK*, is the name of a COMMON block in the program.

F.5.5 Global Table

The **global table** follows the section index table in the object module format. You will recall that the global information header contains a pointer to the global table. Like the section index table, the global table is an array of entries. Each global, like each section, has its own entry. Each entry in the table is 54 bytes in length. Also in the table, global references must precede global definitions. Figure F–6 illustrates the format of a global entry table.



NOTE: The shaded area is currently not used.

Figure F-6. A Global Entry Table

We describe each field below. Use the following example as a reference.

```
*** Globals ***
   5 (  0 00000000) AGAIN
   6 (  2 00000004) TRY
                                                    Marked
```

ID Field

The value in the ID field is used to reference the global in the relocation part of the read/write section templates. The value of the ID must be greater than the number of sections because the ID values (from 1 to the number of sections) refer to sections in the section table. Normally, globals are numbered in the global table sequentially beginning with the number of sections plus one. Like sections, each global has a unique ID. In the example, the globals `AGAIN` and `TRY` are numbered 5 and 6 respectively.

Value Field

The 6-byte value field consists of a 2-byte section ID and a 4-byte section offset. If the global is a reference, this field is 0. In our example, the first global entry, `AGAIN`, is the global reference. Therefore, `AGAIN` contains the ID 0 and the offset `00000000`.

However, if the global is a definition, this field contains the value of the section ID and the offset of the definition. In our example, the second global entry, `TRY`, is the global definition. `TRY` contains the ID 2 and the offset `00000004`. Thus, `TRY` is defined at offset 4 from the start of the DATA$ section.

Old Global Field

The old global field contains either FF (hex), which equals TRUE, or 0, which equals FALSE. If the global is a reference, the field always contains a 0. However, if the global is a definition, the old global field has binder and loader implications. If the global definition is *marked*, the old global field is set to 0. However, if the global definition is *unmarked*, the old global field is set to FF. In our example, the global definition, `TRY`, is marked. Thus, its old global field is set to 0.

The loader and the binder use the old global field for different reasons. The following sections describe the implications of each.

LOADER. The loader uses the old global field to determine whether to enter a global definition in the Known Global Table (KGT) when it installs a library. If the old global field is 0, the loader enters the name and address of the global in the KGT. If the value of the old global field is FF (hex), the loader does not enter the name and address in the KGT. Refer to the Known Global Table section at the end of this chapter for detailed information.

BINDER. The binder's output object module contains all the global definitions of its input object modules. When the binder copies the global definitions to the output object module it changes any old global fields that contain 0 to FF (hex). You can use a number of binder options, such as **-MARK** or **-ALLMARK** to override this behavior. Refer to the *DOMAIN Binder and Librarian Reference* for more information.

When multiple definitions of the same global name occur, the binder uses the old global field to select the global definition. If the input object modules contain multiple definitions of the same global name, the binder resolves all references to the definition with an old global field of 0. If more than one of the multiply defined global entries contains a global field of 0, the binder issues this warning message: `Multiply defined globals` and resolves references to the first definition it encounters. On output object modules, the old global field is always set to FF unless you change it with the binder option **-MARK**.

If none of the global definitions contain a global field that is set to 0 (they all contain FF (hex)), the binder resolves references to the first definition it encounters and does not issue a warning. References resolved in previous bindings remain unchanged. Knowing how the binder treats multiple globals can be useful information when you are binding disparate groups of modules with conflicting global names.

DOMAIN compilers and ASM set the old global field to 0 when they create global definitions in object modules.

> NOTE: The values in the following four fields: *use code*, *type code*, *extent*, and *arglist* are not crucial to the binder's and loader's processing of object modules.

Use Code Field

The 2-byte use code field contains one of the values shown in Table F-3.

Table F-3. Use Code Field Values

Value	Meaning
0	Global defines or references data.
1	Global defines or references a value.
2	Global defines or references a procedure.
3	Global defines or references a function.

DOMAIN compilers set the use field value to the appropriate value.

Type Code Field

DOMAIN compilers currently set this 2-byte field to 1.

Extent Field

Some DOMAIN compilers currently set this 4-byte field to the variable size if the global is the name of the variable; otherwise, they set the field to 0.

Arglist Field

Currently, DOMAIN compilers and the binder set this 4-byte field to 0.

Name Field

The 32-byte name field contains the name of the global. Each global must have a unique name. For more information about names, refer to the "Read-Only Sections" section earlier in this chapter.

F.5.6 Read/Write Section Templates

The **read/write section template** is the last required element of the object module format. However, optional SRI records or module information records (discussed in the "Static Resource Information (SRI) Records" section later in this chapter) can follow the templates. A **read/write section template** consists of a series of records used to initialize the read/write sections. Read/write sections contain static data and instructions that need relocation during loading, or that are modified during execution. Unlike the read-only sections, which are represented by memory image format, read/write sections are represented in template format. Data in the read/write sections are **impure** or **relocatable**.

The zero section attribute causes the loader to set the contents of the section to zero *before* processing the templates. If there is no zero section attribute, the contents of the section are indeterminate before processing the templates. Note that the templates may not initialize the entire section.

Recall that the pointer to impure data of the object module header is a file pointer to the start of the read/write section templates. The templates consist of a series of **text records**, which in turn, can be augmented by **relocation records** and/or a **repeat record**. An **end record** marks the end of the read/write section templates. We discuss each of these records below.

Text Records

Text records, which can vary in size from 12 to 522 bytes, contain the initial data plus the address and size of the initialized area. Each record is the start of a sequence that contains a text record and an optional relocation record and an optional repeat record. Figure F-7 shows the fields in a text record.

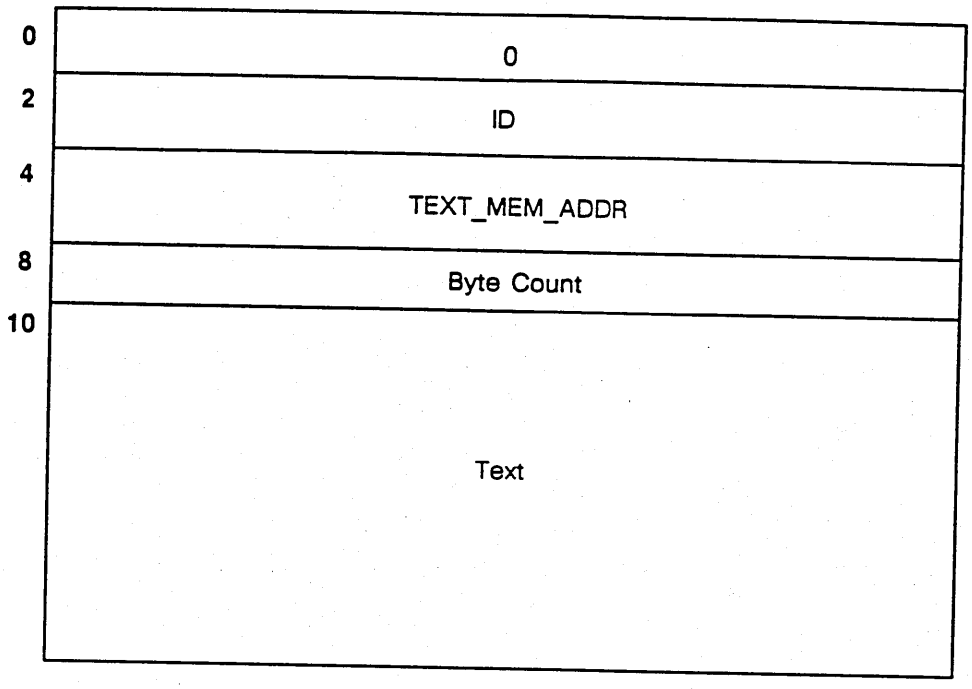


Figure F-7. A Text Record

We describe each field below. Use the following example for reference.

```
*** Impure Data ***
   Text_Rec:
      Id:  2 Text_Mem_Addr: 00000000 Byte_Count:    20
       0000: 00000000 4EF90000 00000000 00140002
       0010: 00000000
   Reloc_Rec:   4
          OFFSET   RELOC_BASE_ID
        1   0000      5
        2   0006      1
        3   000A      2
        4   0010      3

   Text_Rec:
      Id:  4 Text_Mem_Addr: 00000000 Byte_Count:     4
       0000: 000003E8
   Repeat_Rec:        20
   End_Rec
```

0 VALUE FIELD. This 2-byte field contains the value 0 to identify the record as a text record.

ID FIELD. This 2-byte field contains the ID of the section to initialize. The ID of a section is the ordinal position of the field's entry in the section table. In our example, 2 refers to the DATAS section. Refer to the "Section Index Table" section in this chapter for more information.

TEXT_MEM_ADDR FIELD. This 4-byte field contains the byte displacement from the section's beginning. The ID plus the offset give the location to initialize with text. This field is identified as `Text_Mem_Addr: 00000000` in our example, which means the start of the DATAS section.

BYTE COUNT FIELD. This 2-byte field contains the size of the text field that follows. Legal values are between 1 and 512. In our example, the byte count is 20. If the byte count is odd, the assembler, or creator of the object module, appends a filler byte to the text record to start the next record on an even boundary.

TEXT FIELD. The text field contains the initial data. The size of the field can vary from 1 to 512 bytes. The size is set in the byte count field described above.

Relocation Record

A relocation record is an optional augmentation to the previous text record. The loader performs **relocation** on 4-byte quantities aligned on even-byte addresses. The size of a relocation record can vary from 8 to 518 bytes. Our example has four entries. You can have as many as 128 relocation entries (maximum text size = 512 bytes, or 128 4-byte quantities). Figure F-8 illustrates the format of four relocation entries.

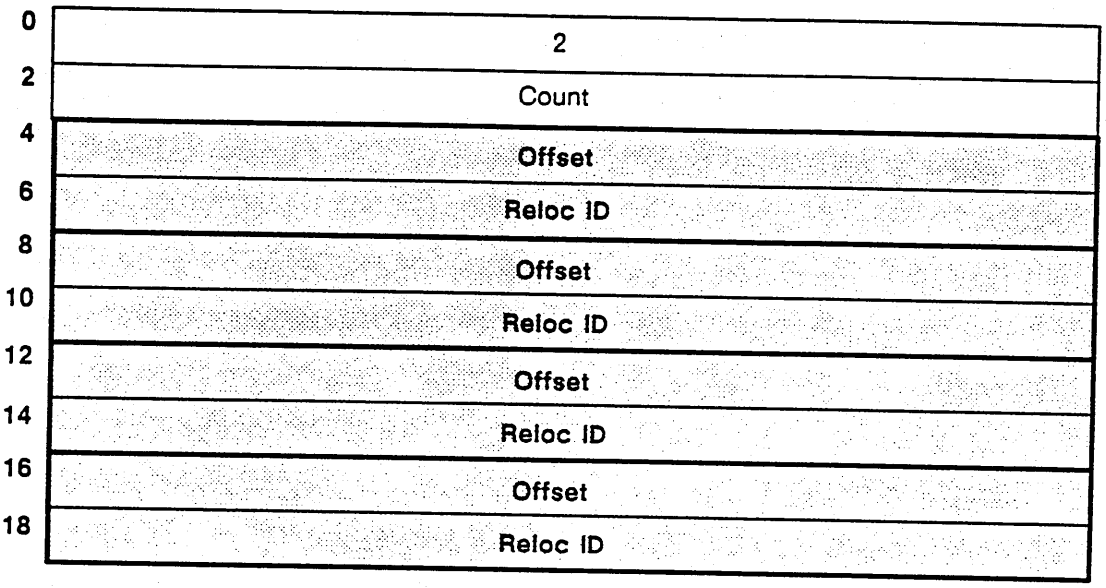


Figure F-8. A Relocation Record with Four Entries

NOTE: The figure above illustrates a relocation record with four entries. Each bordered entry constitutes one entry, which contains an *offset* and a *reloc ID* field.

We describe the fields below. Use the following example for reference.

```
Reloc_Rec:   4
        OFFSET    RELOC_BASE_ID
   1     0000        5        *Global Reference AGAIN
   2     0006        1        *Section PROCEDURE$
   3     000A        2        *Section DATA$
   4     0010        3        *Section DEBUG$
```

2 VALUE FIELD. This 2-byte field contains the value 2, which identifies the record as a relocation record.

COUNT FIELD. This 2-byte field contains the number of relocation entries in this record. The count field value can vary from 1 to 128. In our example, the count value is 4.

OFFSET FIELD. This 2-byte field provides the location of the 4-byte quantity to relocate. The offset is relative to the start of the text record.

RELOC_BASE_ID FIELD. This 2-byte field contains the ID of a section or a global reference. When the loader processes the relocation entry, it adds the address of the section or global to the location identified by the offset.

Repeat Record

The 6-byte repeat record is an optional augmentation of the previous text record. A repeat record repeats the previous record *n* times (repeat count). For example, in the following FORTRAN *DATA* statement, we initialize each element of a 20-element array called `iarray` to 1000:

```
DATA iarray/20 * 1000/
```

Figure F-9 illustrates the format of a repeat record. Use the example that follows the figure as a reference.

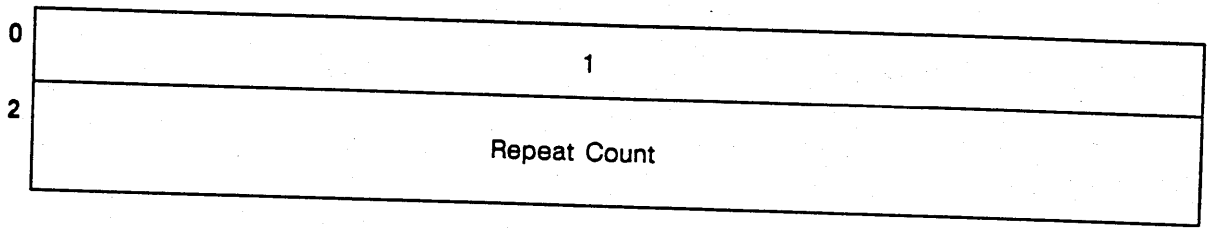


Figure F-9. A Repeat Record

```
Text_Rec:
      Id:  4 Text_Mem_Addr: 00000000 Byte_Count:     4
       0000: 000003E8        [3E8 = 1000 decimal]
  Repeat_Rec:         20
```

1 VALUE FIELD. This 2-byte field contains the value 1, which identifies the record as a repeat record.

REPEAT COUNT FIELD. This 4-byte field contains the number of times to repeat the initialization specified in the previous text record. In the above example, 80 bytes (4*20) are initialized starting at `Id: 4 Text_Mem_Addr: 00000000`.

End Record

The end record marks the end of the read/write section templates. There is only one end record for each object module. Figure F-10 illustrates the format of an end record.

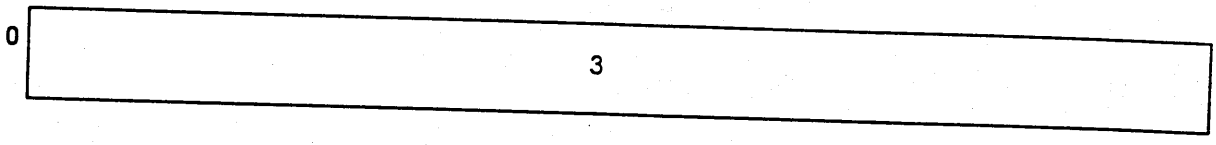


Figure F-10. An End Record

4 VALUE FIELD. This 2-byte field contains the value 3, which identifies the record as an end record.

F.6 Optional Elements of the Object Module

There are a few optional elements that are important in fully understanding object modules. These elements are:

- Module information records (MIR)
- Static Resource Information (SRI) records
- Debug tables

We have already referred to these elements in our discussions within the chapter. Here we explain them in detail.

F.6.1 Module Information Records (MIR)

Module information records supply additional optional information about the object module. Currently, two types of MIRs are defined: a **maker record**, which contains module information, and **object file module information record**, which we discuss on the following page. As shown in Figure F–11, module information begins with a header containing the number of module information records. In the illustration, two individual module information records follow the header. You will recall that the object module header contains a pointer to the module information.

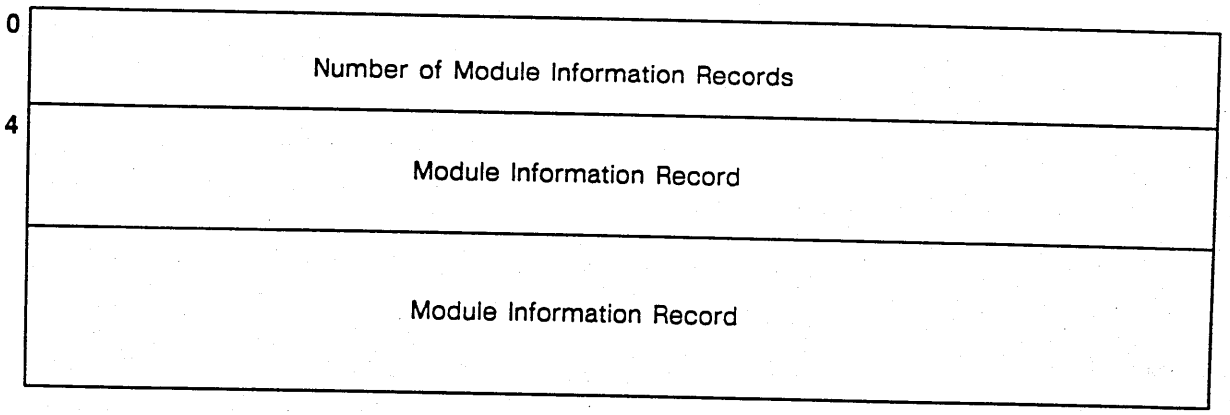


Figure F–11. Module Information Header (with two records)

We describe the field and records below.

Number of Module Information Records Header Field
This 4–byte header field contains the number of records following it. Each record type can have a varying format; however, all records begin with a 2–byte ID field and a 2–byte size field.

Maker Version Module Information Record
The format of the maker version module information record is shown in Figure F–12.

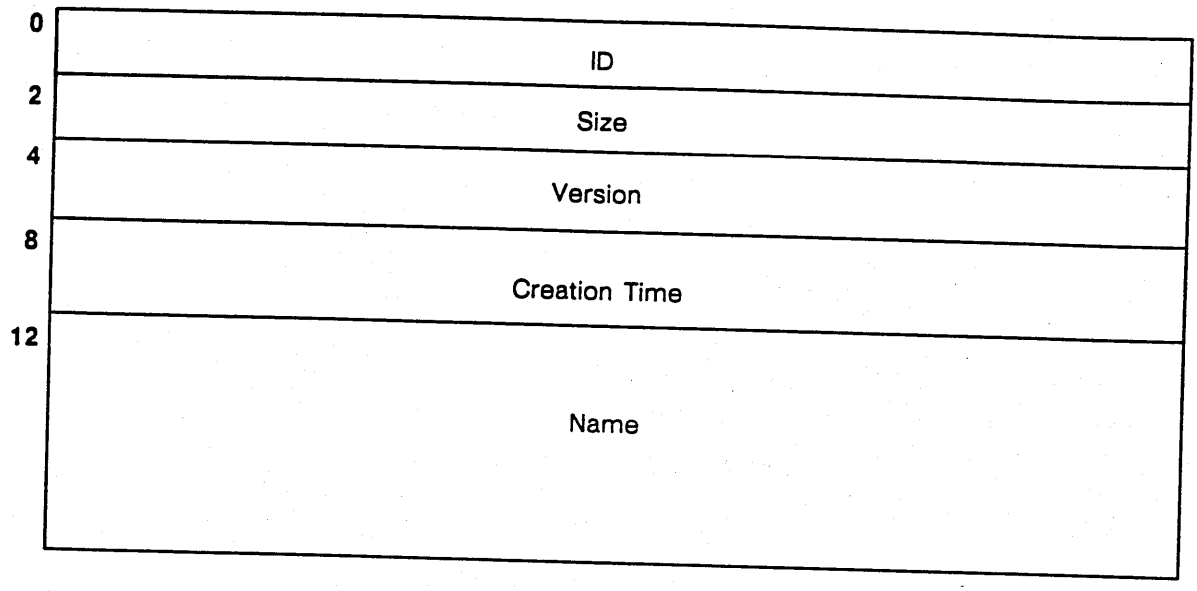


Figure F–12. A Maker Version Module Information Record

We explain each field below.

ID FIELD. This 2–byte field identifies the object module record's type. An ID of 1 identifies a Maker record.

SIZE FIELD. This 2–byte field identifies the byte size of the record including the identification and size field. The byte size of a Maker record is 44.

VERSION FIELD. This field contains the version number displayed at the end of the compiler message,

```
No errors, no warnings, compiler Rev nn.mm
```

or displayed in response to the **–VERSION** option following the compiler invocation command.

The version number displays in an *nn.mm* format. The number is encoded into the 4–byte version field by storing *nn* in the first two bytes and *mm* in the last two bytes. The binder displays the the version field, the creation time field, and the name field from the module information record if you use the **–MAKER** option when you bind your input modules. Note that you can use the bind command to display this information for any object module, for example

```
$ bind  test.bin -maker

This object was made by the following:
   ftn, Rev 9.04, Date: 1987/01/28 10:01:36 EDT (Wed)
All Globals are resolved.
```

CREATION TIME FIELD. This 4–byte field identifies the creation time of the *Maker*. Note that this is *not* the creation time of the object module.

NAME FIELD. This 32–byte field identifies the command name of the *Maker* such as, *ftn, pas,* etc. The field is blank filled if the name has fewer than 32 characters.

Object File Module Information Record

A second type of MIR is called the **object file module information record.** The record identifies the object file for a module in a library file. Figure F–13 illustrates the format of the object file module information record.

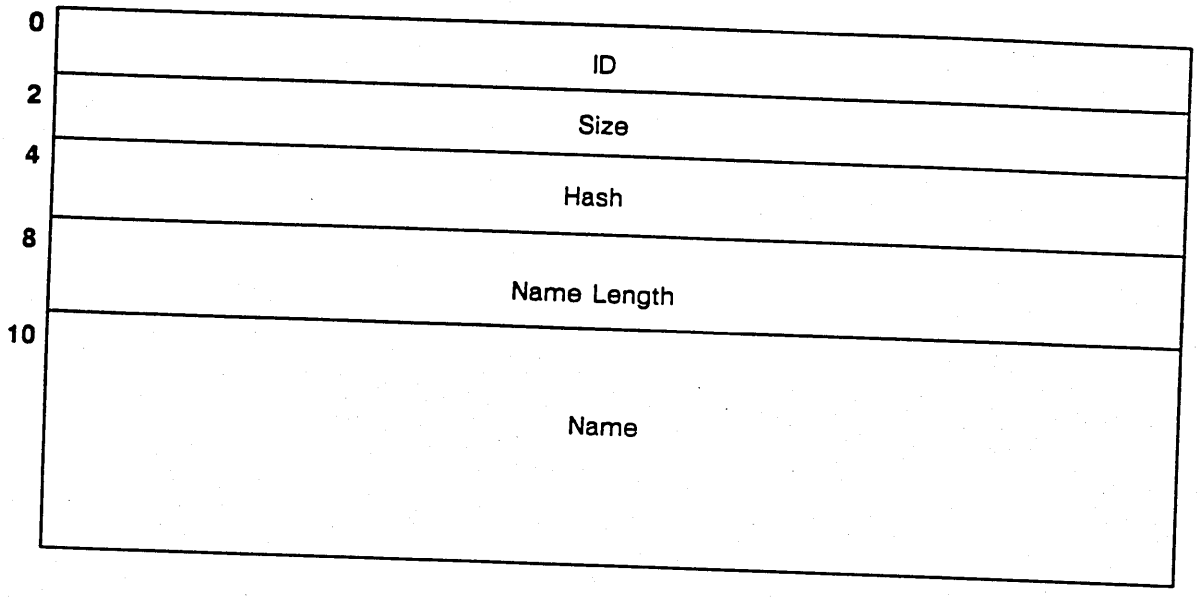


Figure F–13. An Object File Module Information Record

We explain each field below.

ID FIELD. This 2–byte field identifies the kind of object module record. An ID of 2 identifies an Object File MIR.

SIZE FIELD. This 2–byte field identifies the byte size of the record including the identification and size field. The byte size of an Object File record is 42.

HASH FIELD. This 4–byte field contains the hash value of the name.

NAME LENGTH FIELD. This 2–byte field contains the length of the name.

NAME FIELD. This 32–byte field identifies the name of the Object File. The field is blank filled if the name has fewer than 32 characters.

F.6.2 Static Resource Information (SRI) Records

Static resource information (SRI) records are a series of optional 8–byte records that mark object modules with special resource requirements. For example, a program compiled to use a floating–point coprocessor will record this special requirement in an SRI record.

If the object module contains SRI records, the global header information contains a pointer to the first SRI record and contains the number of SRI records that follow contiguously. Figure F–14 illustrates the format of an SRI record.

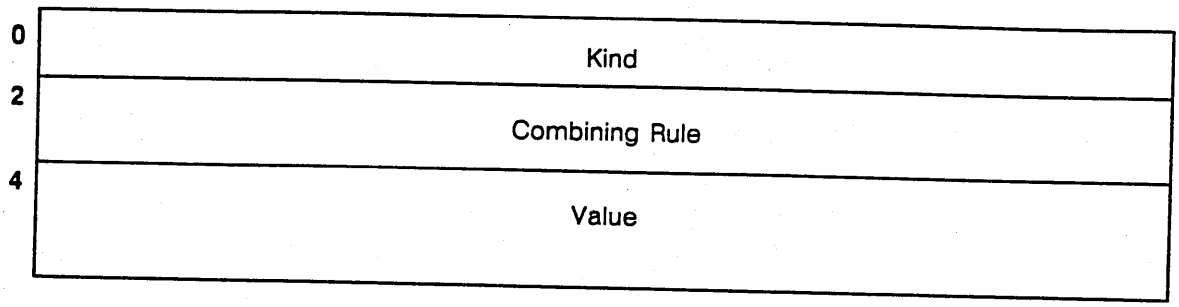


Figure F–14. An SRI Record

We discuss each field below.

Kind Field

This 2–byte field distinguishes different types of SRI records. The three kinds of SRI records (with values) are: *hardware SRI* (1), *software SRI* (2), and *UNIX Version Number SRI* (3). The binder combines SRI records of the same kind according to the combining rule field.

Combining Rule Field

The binder uses this 2–byte field to combine SRI records. The binder combines SRI records with the same **kind** field. Table F–4 lists the values of each combining rule and the rule's definition. Following the value and rule, we provide a detailed description of how the binder interacts with the values from the **kind** field and the combining rule field.

Table F–4. Binder's Interaction with Combining Rule

Value	Combining Rule	Binder's Interaction
0	Take All	Does not combine records. Output obj. module contains multiple SRI records with same Kind field.
1	Take Sum	Combines SRIs with previous record having the same Kind field by adding the two values.
2	Take Max	Combines SRI with previous record and produces an SRI whose value is the maximum of the value fields combined.
3	Take Min	Combines SRI with previous record and produces an SRI whose value is the minimum of the value fields combined.
4	Take Or	Combines SRIs by performing a logical OR on values.
5	Take First	Combines SRIs by setting value to first encountered record.
6	Take Last	Combines SRIs by setting value to last encountered record.
7	Take Unique	Like Take All, except binder issues a warning and discards an SRI with alikekind and value fields.
8	Take Special	Indicates that special binder code is needed to combine SRIs.

Value Field

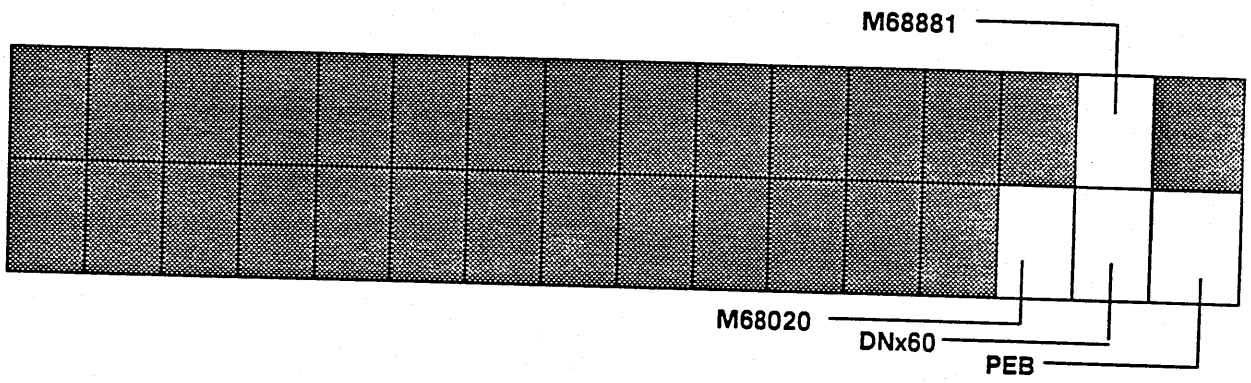
The interpretation of this 4-byte field depends on the value of the Kind field. The meaning of the value field depends on the kind of SRI it is. The three kinds of SRI records are:

- Hardware SRI
- Software SRI
- DOMAIN/IX Version Number SRI

We describe these SRI records below.

HARDWARE SRI. DOMAIN compilers put a hardware SRI record in the object module when the object module execution requires a particular model of node. Normally, DOMAIN compilers generate code that runs on any node model. However, when you use the **-CPU** option (described in Chapter 7) to use a particular model's features, the loader verifies that the node contains the required hardware before invoking the object module.

The value of the kind field of the hardware SRI is 1. The combining rule is Take Or (4). Figure F-15 illustrates the value field.



NOTE: The bits within the shaded area are unused.

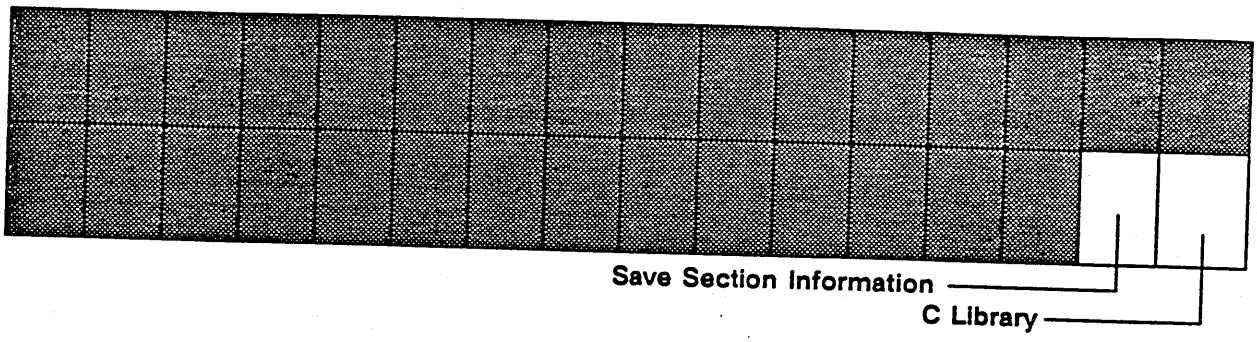
Figure F-15. Hardware SRI Value Field

If the PEB (Performance Enhancement Board) bit is set to 1, the object module requires the enhancement board (for example, DN320) for execution.

If the DNx60 bit is set to 1, the object module requires a DN460, DN660, or DSP160 for execution. If the M68020 bit is set to 1, the object module requires a Motorola 68020 microprocessor. If the M68881 bit is set to 1, the object module requires a Motorola 68881 floating-point coprocessor. The DN330, DN560, and DSP90 have both the M68020 and the M68881.

SOFTWARE SRI. DOMAIN compilers use the software SRI record in the object module to flag the loader to perform special action during load time.

The value of the **kind** field of the software SRI is 2. The combining rule is Take Or (4). Figure F-16 illustrates the value field.



NOTE: The bits within the shaded area are unused.

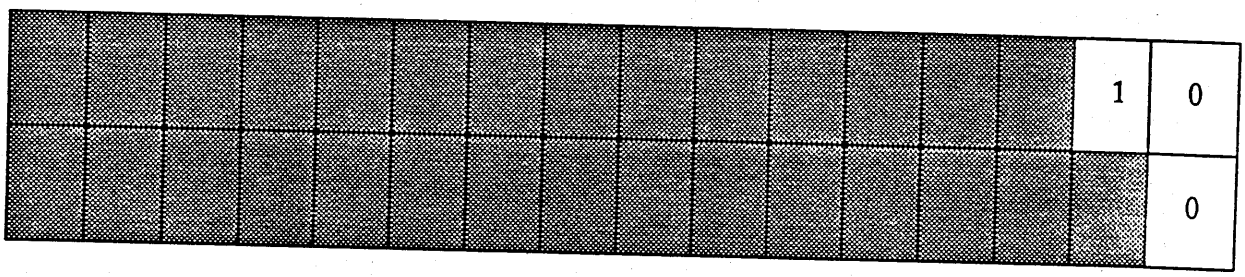
Figure F–16. Software SRI Value Field

The C compiler generates object modules with software SRIs and sets the C library bit to 1. Then, the loader causes C library initialization before executing the object module. The LISP compiler generates object modules with software SRIs and sets the Save Section Information bit. The loader then saves some information when loading an object module.

DOMAIN/IX VERSION NUMBER SRI. Object modules that require a version of DOMAIN/IX use this SRI record, with a **kind** field of 3, to mark the proper version of DOMAIN/IX required for the execution of the object module. This SRI record is necessary because DOMAIN supports multiple versions of DOMAIN/IX, which can have system calls with the same name but different semantics or calling sequences.

The combining rule for a DOMAIN/IX SRI record is **take special** (8). **Take special** means that there is special code in *bind* to handle DOMAIN/IX SRI records. *Bind* flags an error when it attempts to combine two DOMAIN/IX SRI records with different value fields unless one record (or both) has a value field meaning *any version*, as shown in Figure F–17. Refer to the description of the **–SYSTYPE** option in the *DOMAIN C Language Reference* and in the *DOMAIN Binder and Librarian Reference* for more information.

Figures F–17 through F–21 illustrate the five settings of the value field of a DOMAIN/IX SRI record.



NOTE: The bits within the shaded area are unused.

Figure F–17. Value Field of DOMAIN/IX SRI record (runs on any version of DOMAIN/IX).

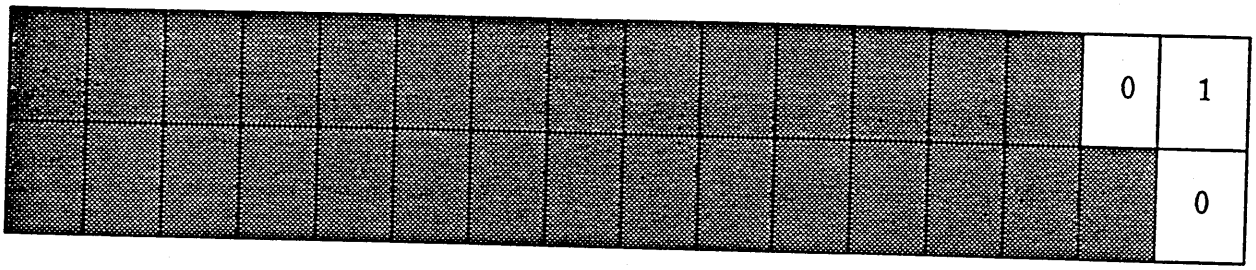


Figure F–18. Value Field of DOMAIN/IX SRI record indicating object module requires DOMAIN/IX version 4.1 BSD.

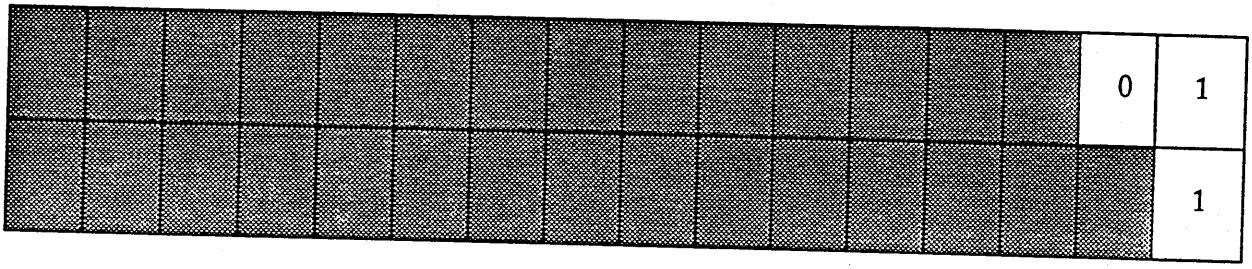


Figure F–19. Value Field of DOMAIN/IX SRI record (requires DOMAIN/IX version 4.2 BSD).

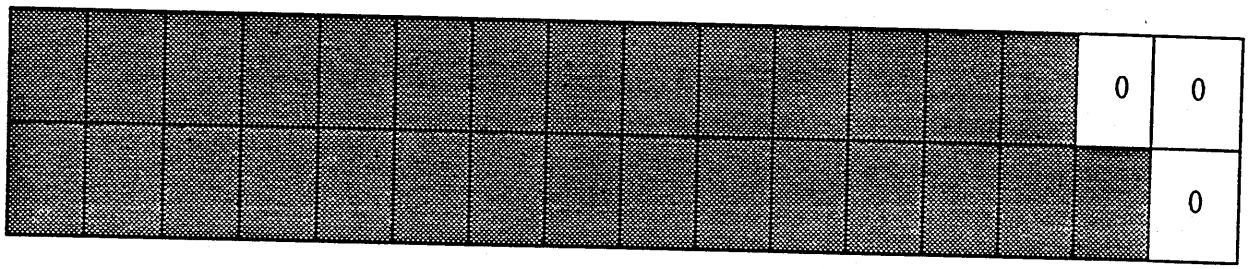


Figure F–20. Value Field of DOMAIN/IX SRI (requires DOMAIN/IX System III).

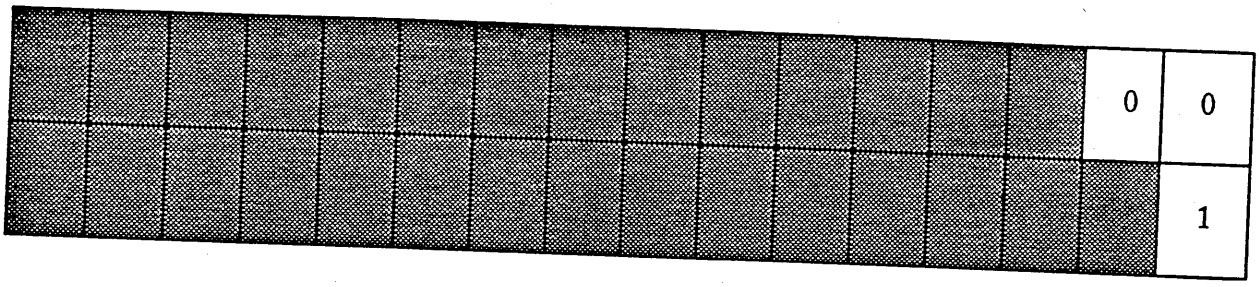


Figure F-21. Value Field of DOMAIN/IX SRI record (requires DOMAIN/IX System V).

F.6.3 Debugging Information

Debugging information is stored in two distinct places in the object module:

- DEBUG$ Section
- Debug Index Table

Each is discussed below.

DEBUG$

DEBUG$ is an optional read-only section that contains debugging information for each procedure or function defined in the object module. The amount of information present for a routine depends on the options under which it was compiled:

-DB	The debugging information stored for a routine compiled with the -DB option (the default) includes the name of the routine, a line number table that relate code locations to source file numbers, and the number and general type of the arguments.
-DBS	These options add a symbol table describing names, types, and locations of -DBA all variables, plus information about the source language and file, structural relations among routines, etc.
-NDB	No information is a present in DEBUG$ for routines compiled with this option. If all routines are compiled with the **-NDB** option, there is no DEBUG$ section.

Note that because DEBUG$ is a section, it is loaded along with the rest of the program. The section name DEBUG$ is treated as special by debuggers and other programs.

The DEBUG$ section contains no information about the actual location of routines. Relating code location to the associated DEBUG$ entry is the primary function of the debug index table. We discuss the debug index table next.

Debug Index Table

The **debug index table** consists of a set of code offset and debug offset pairs that relate the object module offset at the beginning of the routine's pure code to the object module offset of the corresponding DEBUG$ information. The table entries also contain some additional information, such as registers saved by the routine. The format and content of a debug index table entry is described in detail below.

All compiled routines have a debug index table entry, even if they were compiled under the **-NDB** option and thus have no DEBUG$ information. The debug tables are composed of a debug header record and one or more debug entry records. Note that the debug index table is not in a section and therefore is not loaded with the program.

Debug Header Record

The Debug Tables field in the object module header points to the **debug header record**. The debug header record contains the format version and the number of debug entries that follow it. Figure F–21 illustrates the format of the debug header record.

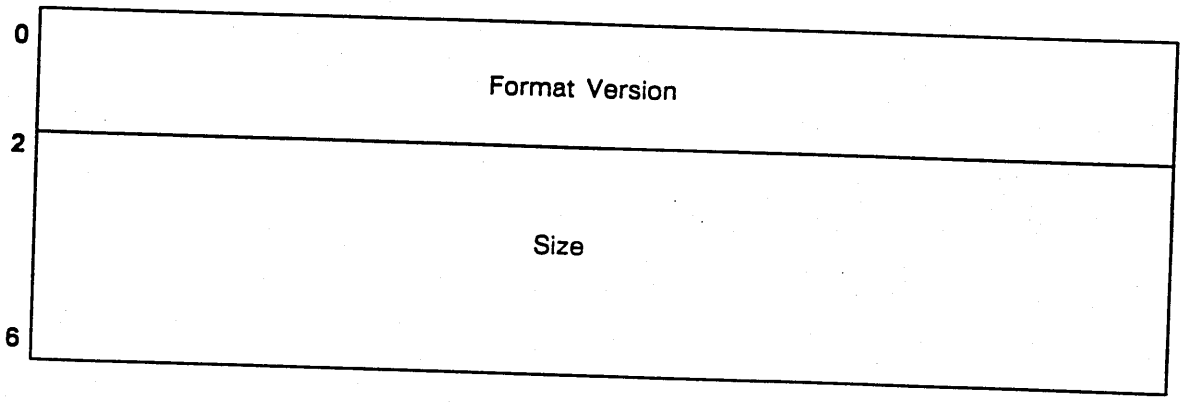


Figure F–22. Debug Header Record.

We discuss each field below.

FORMAT VERSION. The 2–byte format version field identifies the format of the entries that follow it. The current format, described below, is version 1.

SIZE. The 4–byte field contains the number of debug entries that follows it.

Debug Entry Record

The **debug entry record**, which is 16 bytes, relates the PC to the debug tables. Each procedure can have only one debug entry record, and each entry must be located in ascending order of the **code off** field. Note that there should only be one debug entry per procedure, even if the procedure has multiple entry points. Figure F–22 shows the format of a debug entry record.

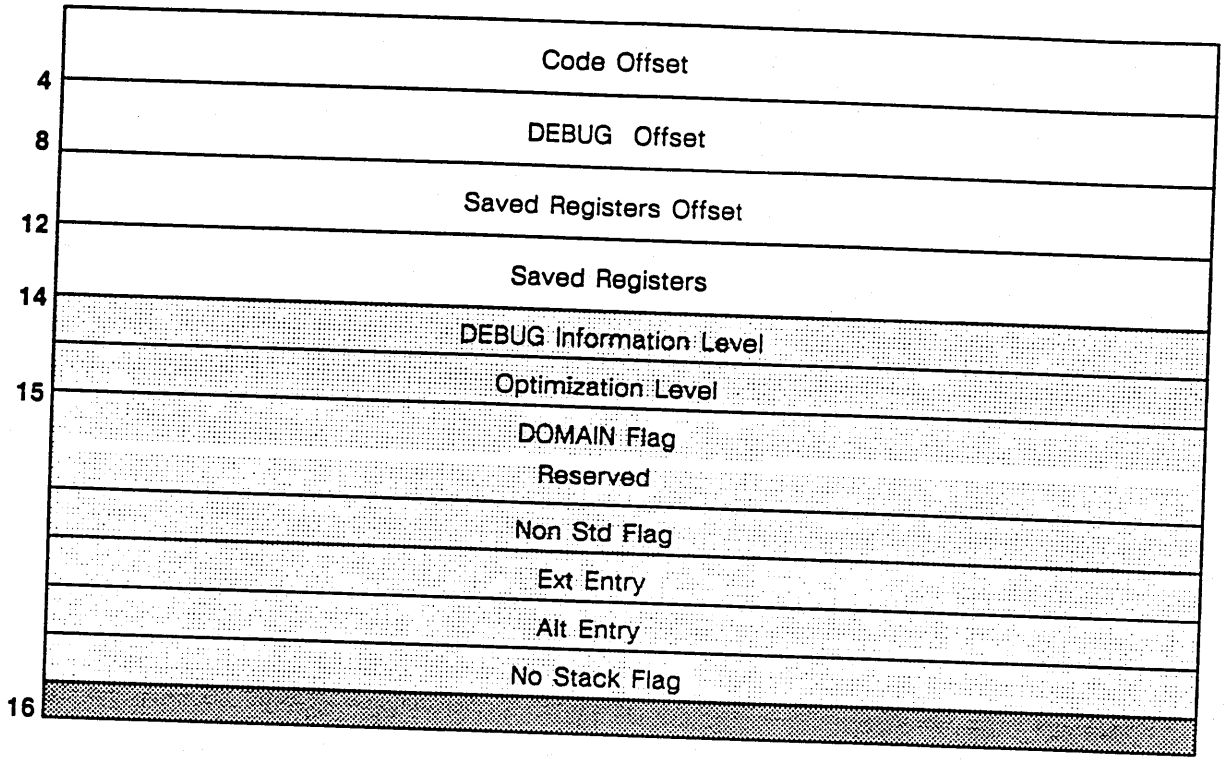


Figure F–23. Debug Entry Record Format

NOTE: The lightly shaded area (between bytes 14 and 16) has been expanded to illustrate the record format. Figure F–24 illustrates the actual format of each of the flags word (2 bytes). The darkly shaded area is currently unused.

We discuss each field below.

CODE OFFSET. The code offset field contains the offset of the start of the code for the procedure from the start of the object module.

DEBUG OFFSET. The debug offset field points to the debug information for the procedure in the DEBUG$ section. The value in this field is set to 0 if you compile with the **-NDB** debugger switch.

SAVED REGISTERS OFFSET. The saved registers offset field is a 32–bit offset of the register save area from the base of the current stack frame.

SAVED REGISTERS. The saved registers field is the set of saved registers. The set consists of A7–A0 and D7–D0. This information enables the debugger to incrementally recover register contents to access variables in up–stack routines.

DEBUG INFORMATION LEVEL. The debug information field level contains a value that codes the level of debugging information available. Table F–5 lists those values.

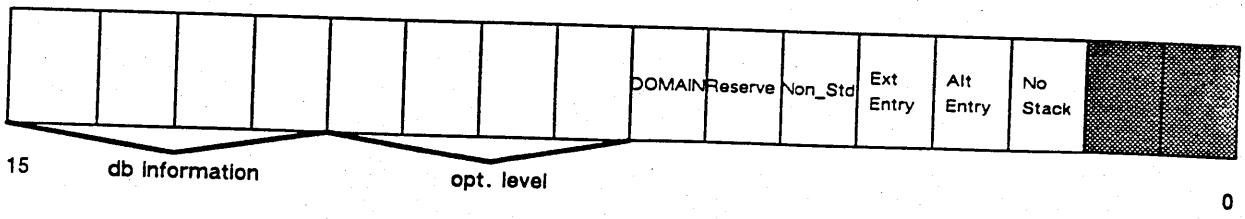


Figure F–24. Format of the Debug Entry Record Flag Word.

Table F-5. Debug Information Level Field Values

Value	Meaning	Debugger Switches/Comments
0	None	-NDB; no debug table; DB OFF set to 0.
1	Name, Line Nos.	-DB option
2	Symbol Table	-DBS and -DBA options
3-15	Reserved	

OPTIMIZATION LEVEL. This field contains the value of the level of optimization. The values are 0-15. Interpretation of the level is compiler-dependent.

DOMAIN FLAG. The value 1 indicates a DOMAIN-written operating system or library routine.

NON STD FLAG. The value 1 indicates that the routine does not use standard stack frame conventions.

EXT ENTRY. The ext entry flag field is true if the program has an external entry prologue (XEP). XEP is the data section code that sets DB (A5) and jumps to the pure code.

ALT ENTRY. The value 1 indicates that the routine has more than one entry point, or an alternate entry point. However, there is only one entry in the debug index for such routines.

NO STACK FLAG. This field indicates that the routine does not use the stack. The flag can only be TRUE if the non std flag is also TRUE.

How Debuggers and Related Programs Locate Debugging Information

Although debugging is not the main topic here, a sketch of the mechanism by which the debugger and related programs, such as TB (TraceBack), locate debugging information, may be of interest. The key factor is that when a program is loaded, its pure code is mapped, *not copied*, into a process address space. Given a virtual address, the operating system is able to return the identity of the file currently mapped there and the offset of the address within it. Given this information for a PC address, a debugger can locate and read the currently executing object module, look up the offset in the Debug Index Table, and locate and read the DEBUG$ information for the current routine.

F.7 Notes on the Known Global Table (KGT)

The **known global table** (KGT) provides the linkage mechanism between your program and the installed libraries. The KGT maintains a list of globals that are defined in the installed libraries. When you load an object module that references data from an installed library or calls an installed library routine, the loader checks the KGT to resolve the reference. The old global field in the global table determines whether to enter a global in the KGT. Refer to the "Global Table" section earlier in this chapter for detailed information.

In addition to maintaining a list of globals, the KGT also maintains a list of sections. This enables your programs and installed libraries to share sections. For example, a FORTRAN routine in an installed library and a FORTRAN routine in your program can reference the same *COMMON* block.

Two section attributes control section sharing: the installed section attribute and the look at installed section attribute. Refer to the "Section Index Table" section earlier in this chapter for information about these section attributes.

Index

Primary page references are listed first. Definition pages are in **boldface.** The letter f indicates that the reference includes a figure. The letter t indicates that the reference includes a table. Symbols are listed at the beginning of the index.

Symbols

A

B

C

K

L

M

N

O

P

R

S

T

U

V

W

X