A/UX Programming Languages and Tools

Volume 1

Release 3.0

 Apple Computer, Inc.

© Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

About This Guide

This guide describes the primary A/UX tools to assist in program development and compilation. It describes the C and Fortran-77 programming languages, their accompanying function libraries and archives (including shared libraries), and the utility programs related to C and Fortran program development. Additionally, this guide details the A/UX implementation of the POSIX standard.

If you want to learn about the utility programs, libraries, and related tools that complement the A/UX compilers, you should read *A/UX Programming Languages and Tools,* Volume 2.

Who should use this guide

This guide is intended for programmers and developers. This guide does not serve as a tutorial to help you learn programming skills; rather, it serves as a reference to determine what tools are available in A/UX and how to use them effectively.

What you need to know

To get the most out of this guide, you need to have a good working knowledge of programming practices. This guide assumes that you are conversant with a programming language and with the general process of coding, compiling, testing, debugging, and so forth. A general knowledge of UNIX is also assumed. You need to know the basic skills of using a Macintosh, such as double-clicking to open a file and dragging the mouse to choose a menu command.

What's covered in this guide

This guide describes:

- the A/UX programming environment
- the command syntax for the C compiler (`cc`)
- the C programming language, with implementation notes for Macintosh hardware
- the standard C, math, and object libraries
- shared libraries
- the command syntax for the Fortran compiler (`f77`)
- the Fortran programming language
- `efl`, an extended Fortran language
- other programming-language utilities, such as `lint`, the C program checker; `sdb`, the symbolic debugger; `as` the assembler; `ld`, the link editor; and the Common Object File Format (COFF)
- POSIX and the A/UX POSIX programming environment, for programming in conformance with the IEEE POSIX standard

Where to go for more information

If you want to learn about the utility programs, libraries, and related tools that complement the A/UX compilers, you should read *A/UX Programming Languages and Tools,* Volume 2. If you need more information about the Macintosh interface, see *A/UX Toolbox: Macintosh ROM Interface.* If you would like information about porting applications to A/UX, see the *A/UX Porting Guide.* If you would like more information about the shell programming environment, see *A/UX Shells and Shell Programming.*

How to use this guide

This guide serves as a reference to help you when programming and using these tools. As a reference book, it is not designed to be read from cover to cover. Each chapter is a discrete description of a particular tool or class of tools, therefore you should skip directly to these compact references.

Conventions used in this guide

A/UX guides follow specific conventions. For example, words that require special emphasis appear in specific fonts or font styles. The following sections describe the conventions used in all A/UX guides.

Keys and key combinations

Certain keys on the keyboard have special names. These modifier and character keys, often used in combination with other keys, perform various functions. In this guide, the names of these keys are in Initial Capital letters followed by SMALL CAPITAL letters.

The key names are

CAPS LOCK	DOWN ARROW (↓)	OPTION	SPACE BAR
COMMAND (⌘)	ENTER	RETURN	TAB
CONTROL	ESCAPE	RIGHT ARROW (→)	UP ARROW (↑)
DELETE	LEFT ARROW (←)	SHIFT	

Sometimes you will see two or more names joined by hyphens. The hyphens indicate that you use two or more keys together to perform a specific function. For example,

Press COMMAND-K

means "Hold down the COMMAND key and press the K key."

Terminology

In A/UX guides, a certain term can represent a specific set of actions. For example, the word *enter* indicates that you type a series of characters on the command line and press the RETURN key. The instruction

Enter `ls`

means "Type `ls` and press the RETURN key."

Here is a list of common terms and the corresponding actions you take.

Term	*Action*
Click	Press and then immediately release the mouse button.
Drag	Position the mouse pointer, press and hold down the mouse button while moving the mouse, and then release the mouse button.
Choose	Activate a command in a menu. To choose a command from a pull-down menu, click once on the menu title and, while holding down the mouse button, drag down until the command is highlighted. Then release the mouse button.
Select	Highlight a selectable object by positioning the mouse pointer on the object and clicking.
Type	Type an entry *without* pressing the RETURN key.
Enter	Type the series of characters indicated and press the RETURN key.

The `Courier` font

Throughout A/UX guides, words that you see on the screen or that you must type exactly as shown are in the `Courier` font. For example, suppose you see this instruction:

Type `date` on the command line and press RETURN.

The word `date` is in the `Courier` font to indicate that you must type it. Suppose you then read this explanation:

Once you press RETURN, you'll see something like this:

```
Tues Oct 17 17:04:00 PDT 1989
```

In this case, `Courier` is used to represent exactly what appears on the screen.

All A/UX manual page names are also shown in the `Courier` font. For example, the entry `ls`(1) indicates that `ls` is the name of a manual page in an A/UX reference manual. See "Manual Page Reference Notation" below for more information on A/UX command reference manuals.

Font styles

Italics are used to indicate that a word or set of words is a placeholder for part of a command. For example,

`cat` *filename*

tells you that *filename* is a placeholder for the name of a file you wish to view. If you want to view the contents of a file named `Elvis`, type the word `Elvis` in place of *filename*. In other words, enter

```
cat Elvis
```

New terms appear in **boldface** where they are defined. Boldface is also used for steps in a series of instructions.

A/UX command syntax

A/UX commands follow a specific command syntax. A typical A/UX command gives the command name first, followed by options and arguments. For example, here is the syntax for the `wc` command:

`wc [-l] [-w] [`*directory*`]`...

In this example, `wc` is the command, `-l` and `-w` are options, *directory* is an argument, and the ellipsis (...) indicates that more than one argument can be used. Note that each command element is separated by a space.

The following list gives more information about the elements of an A/UX command.

Element	*Description*
command	The command name.
option	A character or group of characters that modifies the command. Most options have the form -*option,* where *option* is a letter representing an option. Most commands have one or more options.

Element	*Description*
argument	A modification or specification of a command, usually a filename or symbols representing one or more filenames.
[ ]	Brackets used to enclose an optional item—that is, an item that is not essential for execution of the command.
...	Ellipsis used to indicate that more than one argument can be entered.

For example, the `wc` command is used to count lines, words, and characters in a file. Thus, you can enter

```
wc -w Priscilla
```

In this command line, `-w` is the option that instructs the command to count all of the words in the file, and the argument `Priscilla` is the file to be searched.

Manual page reference notation

A/UX Command Reference, A/UX Programmer's Reference, A/UX System Administrator's Reference, X11 Command Reference for A/UX, and *X11 Programmer's Reference for A/UX* contain descriptions of commands, subroutines, and other related information. Such descriptions are known as *manual pages* (often shortened to *man pages*). Manual pages are organized within these references by section numbers. The standard A/UX cross-reference notation is

command (section)

where *command* is the name of the command, file, or other facility; *section* is the number of the section in which the item resides.

- Items followed by section numbers (1M) and (8) are described in *A/UX System Administrator's Reference.*
- Items followed by section numbers (1) and (6) are described in *A/UX Command Reference.*
- Items followed by section numbers (2), (3), (4), and (5) are described in *A/UX Programmer's Reference.*
- Items followed by section number (1X) are described in *X11 Command Reference for A/UX.*
- Items followed by section numbers (3X) and (3Xt) are described in *X11 Programmer's Reference for A/UX.*

For example,

```
cat(1)
```

refers to the command `cat`, which is described in Section 1 of *A/UX Command Reference*.

You can display manual pages on the screen by using the `man` command. For example, enter the command

```
man cat
```

to display the manual page for the `cat` command, including its description, syntax, options, and other pertinent information. To exit, press the SPACE BAR until you see a command prompt, or type `q` at any time to return immediately to your command prompt.

For more information

To find out where you need to go for more information about how to use A/UX, see *Road Map to A/UX*. This guide contains descriptions of each A/UX guide and ordering information for all the guides in the A/UX documentation suite.

1 Overview of the A/UX Programming Environment

This manual describes some of the program development tools provided with the A/UX operating system. The A/UX programming environment is a very powerful application program development environment. Languages and tools that originated on the UNIX® operating system have gradually migrated to numerous other operating systems, so even if you are new to the A/UX operating system, you might well have already used many of these tools.

There are four main kinds of tools that you use to develop application programs under A/UX:

- language compilers, assemblers, and link editors
- function libraries and archives
- program debugging tools
- other development tools

This manual provides detailed information on the first three categories. A summary of other important development tools (such as SCCS and `make`) can be found in the last section of this chapter; for a complete discussion of these tools, see *A/UX Programming Languages and Tools,* Volume 2. Readers should be conversant with the C programming language and with the general process of coding, compiling, testing, debugging, and so forth.

Programming languages and compilers

The A/UX programming environment includes compilers for several programming languages:

`cc`	The standard C compiler.
`f77`	The standard Fortran compiler.
`efl`	An Extended Fortran Language (EFL) compiler.
`c89`	An ANSI C compiler available as a separate product in A/UX Developer's Tools.
MPW C	A C compiler and development environment for Macintosh applications that run on A/UX or the Macintosh Operating System (OS). This is available as a separate product in A/UX Developer's Tools.

In most instances, the C programming language will be your preferred language for writing applications programs. The C language was developed primarily to provide a portable way of implementing the UNIX operating system and its numerous utility programs. Hence, the connections between the language and the operating system are very deep. Many A/UX utility programs are simply slightly repackaged system calls or subroutines. For example, the shell command `sleep` does nothing more than validate its command-line arguments and then call the `sleep` subroutine. Because of this tight connection, it is often a simple matter to translate a shell script into a functionally equivalent (but much faster) C program.

Aspects of the C language and associated libraries are covered in detail in Chapters 3 through 6. The Fortran language, in its various A/UX incarnations, is discussed in Chapters 11 through 13.

The compilation process

There are six main steps a program goes through on its way to becoming executable:

1. editing
2. preprocessing
3. compilation
4. optimization
5. assembly
6. linking

Of these, the first step is under your direct control, while the remainder is generally handled by the `cc` command. You can influence how the command performs these tasks, but the command takes care of most of the detail work for you. The creation of most programs also includes another step not noted above—debugging.

The C compilation process is shown in Figure 1-1. Your source file, in text format, first goes through a preprocessing phase where it can be combined with other text files. The drawing shows an example in which two header files (`file1.h` and `file2.h`) and one source code file (`file.c`) are combined. The resulting source file is then compiled into an *assembly file.* This assembly file is then (usually) optimized. After optimization, the assembler converts the file into machine code. Finally, the link-editor defines addresses for all functions and variables in your program by resolving external references, either between your source files or standard system functions. It finishes the compilation process by producing an executable object file.

The following sections discuss each of these compilation steps in greater detail.

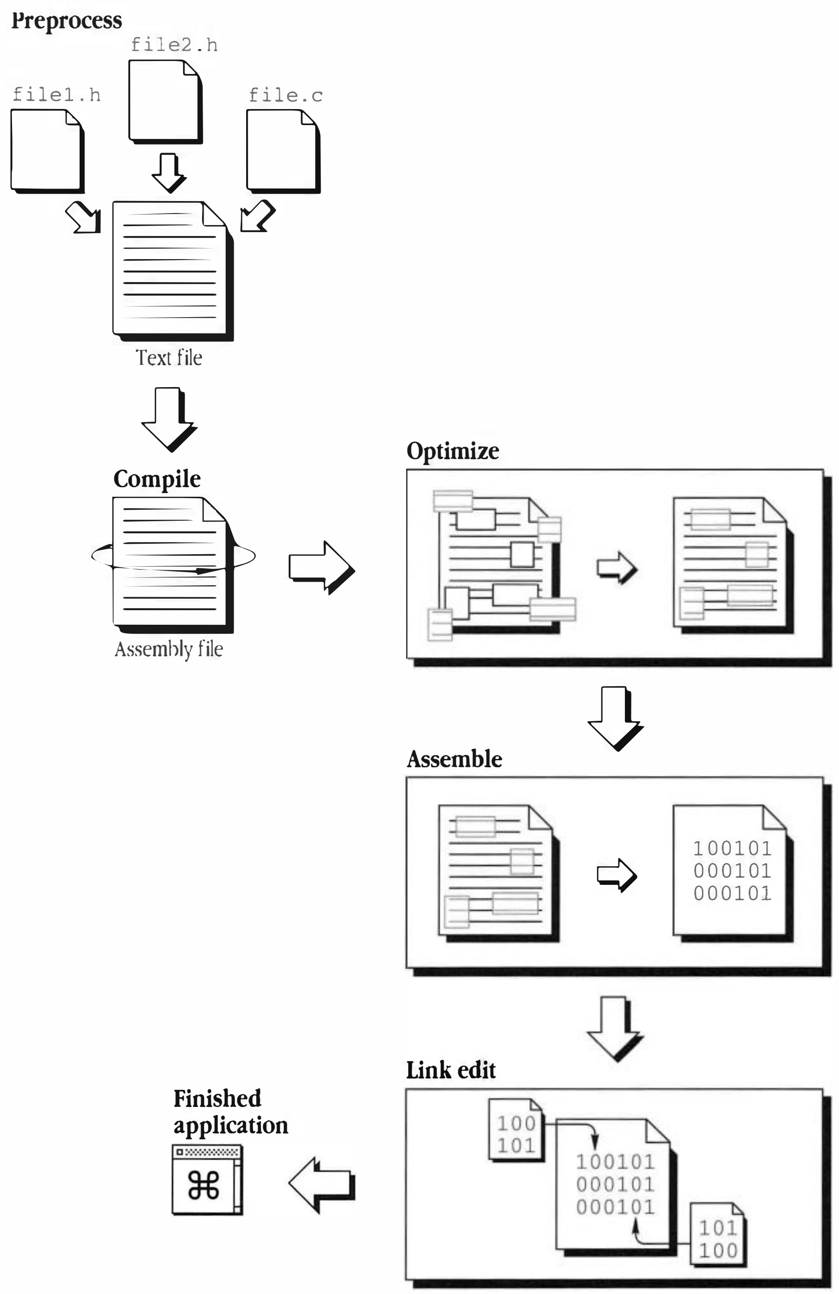

Figure 1-1 Creating a program

Editing

Editing a program is straightforward. With the A/UX operating system, you have the choice of several editors:

1. `TextEditor`. This mouse-based screen editor is unique to A/UX. It is easy to use and intuitive in its operation.
2. `vi`. This screen editor, familiar to many users of UNIX systems, is powerful but requires some perseverance to master.
3. `ed`. This is a UNIX-based line editor, included for those who prefer to use its editing features.

You also can supply other editors of your own choosing, such as `emacs`.

Preprocessing

The preprocessor makes text changes in your source code to get it ready for compilation. These changes include adding specified files into your source code, replacing **macro** text strings, and optionally removing comments. You can include additional sections of code from other files by using an *include directive*. This tells the preprocessor to insert a named file into your source at the location you specify. Figure 1-2 shows the inclusion of a source file at the beginning of a compilation.

You also can have the system insert the declaration for a C library function into your program. These declarations are often made in **header files,** which contain descriptions of the interface to the library routines. By including these header files in your program, the compiler knows how to generate calls to the library routines that are used within your program. The libraries containing the routines must also be specified in the loading phase (this subject is covered more fully in the section "Link Editing," later in this chapter).

In the A/UX C environment, header files are provided that define the functions available for both the standard C library (giving you access to standard UNIX functions) and the Macintosh Toolbox Library (giving you access to Macintosh functions). These files are in the directories `/usr/include` and `/usr/include/mac`, respectively.

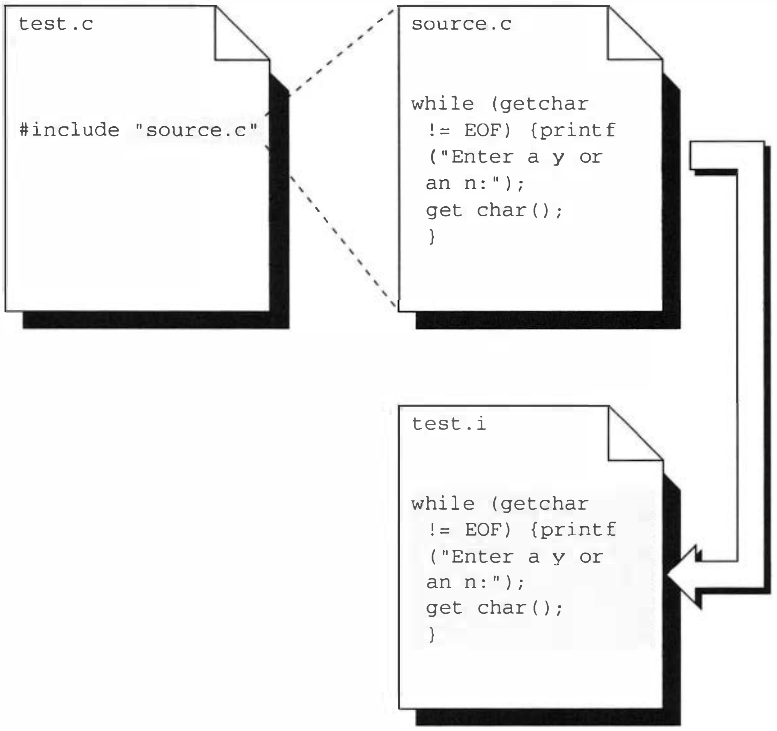

Figure 1-2 Including files

Compilation

The compiler goes through the preprocessed source code once. It groups the characters into tokens and converts them into an internal representation. A **token** is the smallest processing unit used by the compiler; such things as **keywords** (such as `char` and `long`) and **operators** (such as `+` and `&`) are tokens. The internal representations are converted into assembly code. If part of the internal representation cannot be converted into assembly code, the compiler reports an error.

Optimization

Once source code has been compiled into assembly code, the optimizer changes certain instructions to make the code more efficient. This is primarily done by changing the forms of relative address calculation, eliminating unused code, optimizing register allocation, and rewriting local references to use the stack pointer, thus reducing system overhead. The optimizer is a stand-alone program that uses assembly code as both its input and output language. This stage is optional. The optimizer is found in `/lib/optim`.

Assembly

The assembler takes the (optionally) optimized assembly code and converts it into **relocatable** machine-language instructions, placing it in an object file (you call the code at this point **object code**). Relocatable instructions are not bound to a particular memory address. Usually the assembled instructions are for the native processor of the system, though this is not required. The assembler is found in `/bin/as`.

Link editing

The link editor performs several functions to create an executable file:

1. The link editor searches the libraries named on the command line to resolve external references (by default, it also searches a standard set of libraries). An external reference is a call to a function whose code is not in your program. When the link editor finds the machine code that satisfies the external reference, it links the code into your program. This process is shown in Figure 1-3. If the link editor cannot satisfy all external references, it sends you an error message and exits.
2. The link editor binds the machine code to specific memory locations. For example, all global variables are assigned memory addresses so that their values can be accessed.
3. The link editor creates an executable binary Common Object File Format (COFF) file. This process is also called loading.

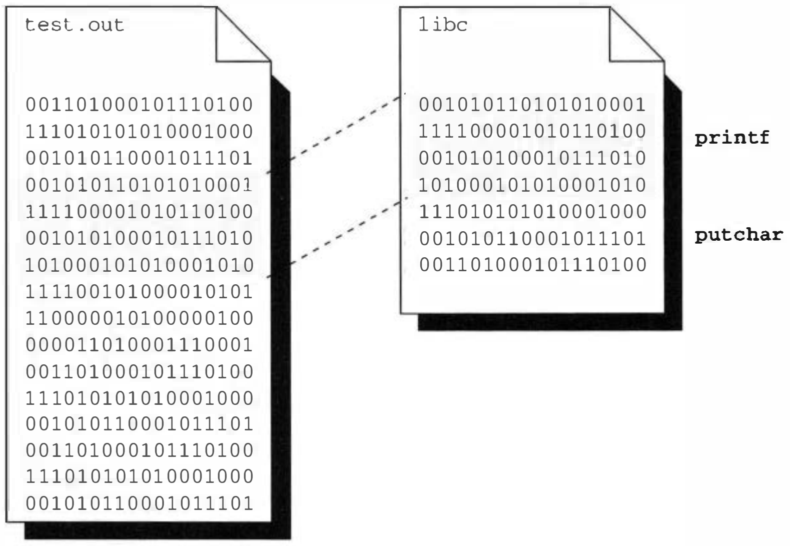

Figure 1-3 Insertion of library code

Other development tools

The following programs for checking and debugging are supported in the A/UX programming environment:

`adb`	`adb` is a tool for debugging A/UX applications at the machine code level. Machine-level debuggers allow you to find the place in the assembly code that is causing problems. This requires more work than symbolic debugging because you must then figure out to which line of source code the faulty assembly instruction corresponds.
`dbx`	`dbx` is a versatile tool that allows you to symbolically debug your A/UX applications at either the source or machine code level. Symbolic debugging allows you to control the program while looking at your source code. A symbolic debugger helps identify the source line that is causing problems. `dbx` also can debug A/UX applications that access the A/UX Toolbox. `dbx` is discussed in detail in Chapter 10.

`lint`	The `lint` program checks C programs for syntax errors, type rule violations, inefficient constructions, potential bugs, inconsistencies, and portability problems. You can specify command line options to instruct `lint` to check only what is necessary for your program. `lint` is discussed in detail in Chapter 8.
`sdb`	The `sdb` program can be used on both C programs and Fortran (`f77`) programs to debug core images or source language after you compile your program using the `-g` option. `sdb` is discussed in detail in Chapter 9.

Libraries and archives

A **library** is a collection of functions and declarations. A **library archive** is a precompiled library whose routines can be linked to other program modules to produce an executable program. It is the job of the link editor (`ld`) to select from a library archive the routines that are necessary to resolve external references in a set of object files.

Code from libraries can be shared; an executable file from the shared library can be used by multiple applications simultaneously. (In contrast, when you use an executable file from a library that is not shared, each application receives a copy.) The shared library often permits more efficient use of system memory than the standard library. A shared library consists of two sublibraries, containing source archives (host library) and executable object files (target library). An executable file from the shared library can be used by multiple applications at the same time. (In contrast, when using an archive executable file that is not shared, each application receives a copy.) The shared library often permits more efficient use of system resources than the standard library. The use of shared libraries is covered in detail in Chapter 7.

Typically, a library archive is indicated by attaching the suffix `.a` to the name of the library. Library archives are usually stored in the system directories `/lib` and `/usr/lib`.

The main C language libraries in the A/UX programming environment are listed here:

`libc`	This is the standard library for C language programs. The C library is made up of functions and declarations used for system calls, file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. It is covered in detail in Chapter 5.
`libc_s`	This is the shared library version of `libc`.
`libm`	This is the mathematical library for C language programs. This library provides exponential, Bessel, logarithmic, hyperbolic, and trigonometric functions. It is covered in detail in Chapter 6.
`libmac`	This is the library for the routines that access the Macintosh Toolbox. For more information, see *A/UX Toolbox: Macintosh ROM Interface*.
`libmac_s`	This is the shared version of `libmac`.
`libld`	This library provides functions for the access and manipulation of COFF files. It is covered in detail in Chapter 6.
`libcurses`	This library provides functions for writing to, reading from, and updating terminal screens. It is covered in detail in *A/UX Programming Languages and Tools,* Volume 2.
`libposix`	This library is for the A/UX POSIX environment. It contains functions that implement the POSIX environment for A/UX. Appendix B discusses this library in greater detail.

There are also several libraries available for use with the `f77` compiler. The following are the most important:

`libF77`	This is the standard Fortran library. It includes various mathematical routines, string functions, and data conversion routines. This library is covered in Chapter 11.
`libI77`	This is the Fortran input/output library. This library is covered in Chapter 11.

In addition, it is also possible to gain access to routines contained in the standard C library, `libc`, from within a Fortran program. All of these libraries are provided in precompiled form only.

The A/UX file system

A/UX supports several different file systems. AT&T UNIX file systems, BSD UNIX file systems, NFS file systems, AppleShare file systems, and Macintosh file systems (HFS) are all available. In A/UX, you can use the Finder to see all file systems.

All file systems supported by A/UX are hierarchical. The hierarchical file system allows a volume to be divided into smaller units known as **directories.** Directories can contain files as well as other directories. The hierarchical structure matches the user's perceived desktop hierarchy in the Finder, where folders contain files or additional folders. In other words, a folder on the desktop is the visual representation of a directory. In all of the file systems that A/UX supports, files are represented by text icons and directories are represented by folder icons.

Structure of the file system

In the A/UX operating system, a **file** is a linear stream of bytes terminated by an end-of-file indicator. No other structure is imposed by the system on a file. This fact makes it extremely straightforward to write programs that perform simple file manipulation. Programs can process data streams one character at a time; there is no need to read or write files according to a fixed-length record format (as in some other operating environments). In addition, because of this simplicity, the system can treat virtually every object it handles (such as input/output data streams) as a file. Even terminal screens and peripherals are dealt with as files.

Files can be placed anywhere (possibly in multiple locations) in a hierarchy of directories. A **directory** is simply a file that you cannot write. It contains the names of the files *in* that directory and an indication of where to find the files on the disk.

In A/UX, a **file system** is a logical device containing the data structures that implement all or part of the directory hierarchy. The **directory hierarchy** is the collection of all files on the currently mounted (accessible) file systems.

File descriptors

To gain access to a file resident in the file system, a process must first open that file. A typical way to open a file is to use the `open` system call. When successful, this call returns a **file descriptor,** an integer that can be used in other system calls and subroutines to refer to the file.

Three files are opened automatically for each user process running under the A/UX operating system: `stdin`, `stdout`, and `stderr`. These are the standard input, the standard output, and the standard error files, and are associated, respectively, with the file descriptors `0`, `1`, and `2`.

Creating and deleting files

The `close` system call closes an open file. To create a new file, you can use the `creat` system call. To remove a file from the file system, you can use the `unlink` system call. To create and remove directories, use `mkdir` and `rmdir`.

Retrieving and changing attributes of files

There are a number of other system calls that allow the programmer to ascertain the status and modify the attributes of files. Among these are `stat`, `chown`, `chmod`, `chdir`, `ulimit`, and `umask`.

Special files

There is a further kind of file in the A/UX operating system, called a **special file.** Special files are contained in the system directory `/dev`. Each file in `/dev` contains the description of a device and is used to associate a device name with a physical device. There are three classes of special files: **block, character,** and **fifo,** each of which requires its own input and output system. All three types of special files, however, are created with the system call `mknod`.

A block device is a collection of random-access memory blocks. It is accessed through a layer of software that caches these blocks in an array of system buffers. When a request occurs to read a block of some device, the buffers are searched to determine whether one of them contains the requested data; if so, the device does not need to be physically accessed, because the contents of the buffer can be supplied instead. Writes are performed in an analogous manner: a buffer is filled with the modified data and the actual block device is not updated until the operating system flushes its buffers. Some reads and most writes are thus asynchronous (see "Asynchronous I/O," later in this chapter).

A character device is anything other than a block device. I/O requests are sent to the driver virtually untouched. It is up to each device driver to determine how to handle a character I/O request. A disk driver, for example, passes the request through untouched and the transfer is directly from or to user space. For a traditional character device, such as communications lines and line printers, the driver buffers the user's I/O requests.

A fifo is a special file that is also referred to as a *named pipe*. Fifos are discussed, along with pipes, in the section "Pipes and fifos," later in this chapter.

Performing input and output

The C language contains numerous facilities for obtaining data from an input stream and for sending data into an output stream.

Formatted I/O

It is possible to read and write files according to a fixed format, when it is necessary or useful to do this. The subroutine `scanf`, for instance, reads data from the standard input file in a format specified by its first argument. Similarly, the routine `printf` puts data on the standard output file in a format specified by its first argument. In either case, it is also possible to read or write files other than the standard input or output. See `scanf`(3S) and `printf`(3S) in *A/UX Programmer's Reference* for details.

Buffered I/O

It is not necessary to perform either input or output in fixed-length records; primitives exist for reading characters (bytes), or words (32-bit integers) from the input and for writing characters or words on the output. See `getc`(3S) and `putc`(3S) *A/UX Programmer's Reference* for details.

File I/O

The A/UX system includes a number of system calls and subroutines for performing low-level input and output. The `open` and `close` system calls, which, respectively, open and close files accessible to programs, have already been mentioned. Associated with the file descriptor returned by a successful `open` call is a pointer into the file called a **file pointer.** This indicates the point at which subsequent reading or writing is to occur. If the `open` call is invoked with the `O_APPEND` flag, for instance, the file pointer is positioned at the end of the file; otherwise it is placed at the beginning.

The two most fundamental file I/O primitives are `read` and `write`. The `read` call copies a specified number of bytes from the current read position in the file (as indicated by the file pointer) into a buffer. Conversely, the `write` call copies a specified number of bytes from a buffer to the current write position in the file (as indicated by the file pointer).

The file pointer is moved automatically whenever a `read` or `write` is performed; it also can be moved explicitly, without performing any actual input or output, with the `lseek` system call. The position in the file to which the file pointer is to be moved can be specified as an offset relative to the beginning of the file, the end of the file, or the current position of the file pointer in the file. In all cases, however, the return value of the `lseek` call is the offset in bytes from the beginning of the file.

Once a file is opened, its status and permissions can be controlled with the `fcntl` system call. For example, parts of the file can be **locked** to prevent either reading or writing those parts of the file. The `fcntl` call also can be used to duplicate file descriptors.

Pipes and fifos

The A/UX operating system supports yet another type of file, called the **pipe.** A pipe is a data stream that must be read in order; that is, there is no random access. Because it is a type of file, a pipe is assigned an inode when it is created; an unnamed pipe, however, in contrast to a named pipe, does not reside in a directory or take up space in the file system. It is a temporary file created by the operating system to pass data between related processes.

Pipes are created by invoking the `pipe` system call. Once created, a pipe can be read or written with the `read` and `write` functions mentioned earlier. There must be a process at each end of the pipe, one writing data and the other reading data. The data passing through a pipe cannot be reread. At most, a single character of data can be put back into the pipe using the subroutine `ungetc`. There are two types of pipes—named and unnamed. Unlike named pipes, unnamed pipes are unidirectional: data can flow in only one direction through them. See `pipe`(2) for details.

A **named pipe** (also called a fifo special file) allows the same sort of exchange of data among processes typified by "unnamed" pipes. Because a named pipe is a special file, it resides in the file system. It is created, like the other special files, with the `mknod` system call. A named pipe is opened with the `open` system call and is read from or written to with the `read` and `write` routines discussed in the next section. Like a pipe, a fifo requires data to be read in the order in which it was written to the file. A named pipe allows data to pass in both directions. More importantly, the processes writing to or reading from the named pipe do not have to be related in any way.

Device control

Output to character special devices can make use of an additional system call, `ioctl`, which is used to perform a variety of device control functions. A computer that contained a built-in speaker, for example, could use `ioctl` to adjust the parameters affecting speaker output, such as volume, pitch, or duration. Similarly, a program could use `ioctl` to eject a floppy disk from the computer. The common element here is that `ioctl` is used to control the device, not to read or write data. See `ioctl`(2) and section 7 of *A/UX System Administrator's Reference* for control commands for a particular device.

Asynchronous I/O

Asynchronous I/O happens most of the time when the I/O is both buffered and blocked. When this happens, `reads` can precede a request, while `writes` lag behind. Historically, the need for anticipatory reading (for faster response to `reads`) led to buffering, while the need to minimize disk access led to blocking.

The definition of block caching (see the paragraph in "Special Files" on block devices, earlier in this chapter) mentions the array of system buffers in which a block device caches blocks of some file. In fact, the system maintains parallel arrays of buffers, consisting of input buffers and output buffers. The input buffers receive the results of `reads`, while the output buffers hold intended `writes`.

When a process requests a `read` call, the system returns the results immediately, synchronously with the request. Thus, `reads` do not appear asynchronous, but can be so. If the data sought already has been cached into an input buffer, there is no need to read the data from disk, as it already was read into the input buffer previously.

The A/UX operating system buffers `write` calls until they are absolutely necessary, because actual disk access is relatively slow. When you ask for a `write` (for instance, while editing a file), the operating system responds with the character count and filename, as if it were writing the file to disk. However, it is actually writing to the output buffer.

A `write` call to disk is forced under the following conditions:

- All memory buffers are full.
- `sync`(2) has been sent, requesting an update of the superblock.
- The system is about to crash, and files must be written to disk to avoid losing them.

Thus the relation in Table 1-1 holds.

Table 1-1 Buffer versus disk access with asynchronous I/O

Process	Buffer access	Disk access
`read`	synchronous	asynchronous
`write`	synchronous	asynchronous

Process control

Processes are created by the `fork` system primitive. The newly created process, called the **child,** is a copy of the original process, called the **parent.** There is no detectable sharing of primary memory between the two processes (though, of course, if the parent process is executing from a read-only text segment, the child shares the text segment). Copies of all writable data segments are made for the child process. Files that were open before the `fork` is called are shared after the `fork` is completed. The processes are informed of their parts in the relationship, allowing them to select their own (usually nonidentical) destiny. The parent can wait for the termination of any of its children. This is accomplished through the `wait` system call.

A process can execute a file through use of the `exec` system call. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. An `exec` does not change processes; the process that performed the `exec` persists, but after performing the `exec` it is executing a different program. Files that were open before the `exec` remain open afterward. The exec system call never returns control to the calling process (except when an error is encountered).

If an executing program (for example, the first pass of a compiler) wants to overlay itself with another program (for example, the second pass), then the executing program simply calls the `exec` system call on the second program. In this sense, an `exec` is analogous to a `goto` statement in the executing program.

If, however, the executing program needs to regain control of execution after it uses the `exec` call on a second program, it should first use the `fork` call to create a child process, have the child use the `exec` call on the second program, and have the parent wait for the child. This is analogous to a subroutine call in the executing program.

A process can terminate by overlaying itself with a new process, as described earlier in connection with the `exec` routines. A more standard way to terminate a process is by invoking the `exit` system call. Invoking the `exit` system call closes all open file descriptors, notifies all parents of the process termination, unlocks all process, text, or data locks currently active, and returns an exit status to the parent process.

Signals

Process execution can be controlled externally to the process by using **signals.** A signal is a software interrupt that usually indicates some exceptional or error condition. The signal `SIGSYS`, for instance, indicates that a bad argument to a system call was detected by the system. See `signal`(3) for a list of signals.

Signals can be sent by the operating system, by the user from the shell, or from another user program; this is accomplished using either the `kill` shell command or the `kill` system call. The program to which the signal is sent can choose one of three ways to respond. The program receiving the signal can ignore the signal, it can terminate upon receipt of the signal, or it can call a function in response to the signal. These options are selected using the `signal` system call. Some signals, however, cannot be caught or ignored. In particular, the `SIGKILL` signal cannot be ignored by the receiving process.

A typical signal-handling scenario is as follows: A process indicates that it will catch designated signals through the `signal` system call. A `signal` call simply associates the address of a process signal-catching routine with the corresponding signals for later use by the system. When such a signal is delivered, the kernel interrupts user-level execution and transfers control to the signal-catching routine. The signal catcher notifies the user process that a signal occurred (for example, through a global flag) and returns to the kernel. The user-level execution resumes where it left off before the signal arrived. Normally, the user process checks the global flag at intervals and, finding that a signal arrived, performs the appropriate processing.

User programs that need to process signals should have a separate signal-catching subroutine that simply sets a global flag of some type and exits. While it is possible to do more in a signal catcher, it is not usually wise to do so, especially in cases where the actions of a signal catcher could interfere with the completion of atomic operations.

The A/UX implementation of signals allows a process to determine which of two methods to use to process signals. A process can interpret signals in accordance with the System V Interface Definition (SVID) or with the conventions of the Berkeley Software Distribution, Release 4.3 (4.3 BSD). The primary difference between the two implementations of signal handling is that Berkeley signals are said to be *reliable,* whereas SVID signals are not. A program handles signals reliably if a signal sent to it is guaranteed to be processed. This means that if a signal is already being handled, any new incoming signals are caught and queued until they can be processed. Using SVID-compatible signals, this is not always the case; in certain circumstances, a program can lose signals, possibly resulting in the premature termination of the program. For more details, see `set42sig`(3) and `setcompat`(2).

In the A/UX POSIX environment, there is a further implementation of signal handling that is based largely on the BSD approach. The POSIX implementation is intended to provide a set of routines that are more portable across operating environments than either the SVID-compatible or BSD-compatible routines. For a brief discussion of POSIX signals, see Appendix B in this volume. More detailed information about POSIX signals and their relation to SVID and BSD signals can be found in the manual pages entries `sigaction`(3P), `sigprocmask`(3P), `sigsetops`(3P), and `sigsuspend`(3P).

Interprocess communication

The type of interaction between independent processes provided by signals is of a rather limited kind. To allow greater flexibility in the interactions between processes, three further types of interprocess communication have been developed: semaphores, message queues, and sockets.

A **semaphore** is simply a positive integer. Semaphores can function as a means of interprocess communication because they are stored in a memory location that is accessible to various programs through certain system calls. By reading the values of semaphores and, if needed, by altering those values, a program can inspect and control the operation of another process or group of processes. Programs can, for example, suspend operation until a particular semaphore attains some value.

A semaphore is created with the `semget` system call and can be incremented or decremented (by any process that has such permissions) through the `semop` system call. Finally, semaphores can be removed and the memory associated with them freed by use of the `semctl` system call. The `semctl` operation is also used to read and set values of semaphores.

A **message** is a discrete portion of data stored in a buffer that is accessible to a number of independent processes. Any number of messages can be available at one time, so they are stored in a structure called a **message queue.** A process can send a message to such a queue, read messages from it, and alter its process of execution according to messages it receives.

A message queue is created with the `msgget` system call. Messages are sent and received with the calls `msgsnd` and `msgrcv`, and message queues are removed with the `msgctl` system call.

The third type of interprocess communication facility, the **socket,** is especially suited for setting up communications networks among different computers and underlies the Berkeley networking software. A socket is an endpoint for communication; different processes, and indeed different computers, can exchange data and messages through sockets. For full details on implementing sockets and programming with them, see *A/UX Network Applications Programming*.

Program pause and wake up

There are several ways to suspend program execution until some external event occurs. As noted, the implementations of both semaphores and message queues allow a process to wait until a particular semaphore or message is received from some other process. A program also can be made to pause until it receives a signal with the `pause` system call. The signal must, of course, be one that has not been set to be ignored by the calling process.

Once a process is suspended with the `pause` system call, it is typically awakened with the `SIGALRM` signal. A process can arrange to send this signal to itself after a specified amount of time by invoking the `alarm` system call. A call of the form `alarm(`*n*`)` instructs the alarm clock of the calling process to send the signal `SIGALRM` to the calling process after *n* seconds. This call does not itself suspend execution of the calling process.

Other process attributes

There are several system calls that allow a process to determine its own process ID, the process ID of its parent process, and its process group ID. See `getpid`(2) for details.

Memory management

Dynamic memory allocation

Managing the available core memory is an important task for A/UX and any operating system that allows multiple simultaneous processes and multiple users. The system must ensure that each process has access to whatever memory it needs, that other processes do not try to gain access to that memory illegally, and that memory is reclaimed when a process exits. The system also might need to allocate additional memory to an executing process. The A/UX environment provides a number of system calls and library routines for managing the memory storage use of a program.

The primary memory allocation request is `malloc`. A successful call of the form `malloc(`*n*`)` returns a pointer to *n* bytes of free memory. Memory can be returned to the operating system by calling the `free` routine. Other available memory allocation routines include `realloc`, `calloc`, and `cfree`. For an explanation of these routines, see `malloc`(3C) and Chapter 5, "The Standard C Library (`libc`)."

These standard memory allocation routines are designed to be space-efficient, sacrificing speed for smaller data space and code size. There is an alternate set of memory allocation routines that is designed to run considerably faster than the standard set of routines, though at the cost of increased code size and increased memory usage. You can use these time-efficient versions of `malloc`, `free`, and so forth, by using the `-lmalloc` option to the compiler. See `cc`(1) and `malloc`(3X).

Shared memory

There is another form of interprocess communication available under the A/UX operating system called **shared memory.** Using this facility, a process can arrange to share a core memory data segment with other processes, thereby allowing a very fast means for two or more independent processes to share data. This can be useful for applications such as database management or multiplayer games where several independent processes need to inspect (or modify) a common data segment.

A shared data segment of memory is created using the `shmget` system call. Other processes can then gain access to this segment of memory, provided that they possess permissions specified at the time the segment was created. A process can attach itself to a shared segment of memory by invoking the `shmat` system call and detach itself from that segment by invoking the `shmdt` system call. A shared memory segment is removed by using the `shmctl` system call; this call may also be used to alter the permissions associated with the memory segment and to perform other operations on the segment (such as locking it into core memory). For further details on shared memory, see `shmget`(2), `shmctl`(2), and `shmop`(2).

The environment

Whenever a program begins running, the operating system makes available to it the set of all data inherited from the parent process. This set of data is called the **environment** and includes an array of strings as well as information from the parent process such as the user identification (UID), group identification (GID), current directory, and so on. The program can read the strings it finds in the environment and modify its subsequent actions according to the results it receives. A program also can change the strings or add further strings to the environment. By convention, the strings in the environment are of the form

name=value

The environment that each process inherits includes the names `HOME`, `PATH`, `SHELL`, `TERM`, and others. A program can read the environment by executing a call of the form `getenv` (*name*). It can alter the environment it receives from the shell by executing a call of the form `putenv` (*string*), where *string* is of the preceding form.

It is a general characteristic of the A/UX operating system that a process can change its own environment (and the environment of any subprocesses it creates) but not that of its parent process. So, a call to `putenv` affects only the environment of the calling process and of all processes that the calling process creates. Changes made to the environment do not persist after that process has exited. For further information, refer to `putenv`(3C) and `environ`(5).

Using shell commands

It is possible to execute an arbitrary shell command from within a C program by using the `system` subroutine. A call of the form `system` (*string*) results in the program passing *string* to an instance of `/bin/sh` for execution, exactly as if *string* was typed to the shell during an interactive login session. For instance, if a program detects that a certain file needs to be time-stamped, it can accomplish this by calling the function

```
system("touch /usr/tmp/dungeons")
```

The `system` subroutine makes no provisions for capturing any output produced by the executing command. It is possible to send output to a file by including standard shell redirection metacharacters in the argument string, but the file thereby created must then be opened and read if the data stored there is to be accessible to the original program.

A better way to get access to the output of a shell command is to use the `popen` subroutine. The form of the `popen` function is

`popen(`*string*`,` *mode*`)`

where *string* is exactly like the single argument to `system` and *mode* is either `r` or `w`, indicating that the calling program is to read from or write to the specified command. A successful call to `popen` returns a pointer to a file stream that can be used in subsequent reads or writes. See `popen`(3S) for further details.

It is also possible to process command line arguments from within a C program by using the `getopt` subroutine. See `getopt`(3C) for details and an example.

Error handling

The C language interface to the A/UX operating system provides a general facility for detecting and reporting error conditions that can arise from invoking many of the system calls and subroutines discussed above. When a system call returns, it typically returns an integer value to its calling process. A successful function call usually returns a value of 0. Some calls, however, return a nonzero, positive value; for instance, a successful `open` call returns a non-negative integer that is the file descriptor of the opened file.

An unsuccessful system call returns a value of -1. To provide the calling program with a general and automatic way of further specifying the cause of the error, the system maintains a global variable, `errno`, which is automatically set to a nonzero positive value indicating the cause of the error. Thus, every unsuccessful system call results in the following two actions:

- A return value of -1 is returned to the calling program.
- The global variable `errno` is set to some positive integer.

When the program detects an unsuccessful call by inspecting its return value, it can further inspect the value of `errno` to determine the precise cause of failure. Note that `errno` is not reset by successful system calls, so it is important to inspect its value only after an unsuccessful system call.

A program can report the occurrence of an error by using the `perror` subroutine. `perror` prints a message on the standard error output file that describes the last error received by a system call. The printed message consists of two parts: first, the argument (if any) provided to the call to `perror` is printed, followed by a colon, a space, and an indication of the precise nature of the error. `perror` determines the nature of the error by inspecting the `errno` variable.

It is the responsibility of the calling program to detect and react to error conditions indicated by unsuccessful function calls. In addition to the `errno` variable and the `perror` subroutine, the A/UX system also provides an array, `sys_errlist`, containing the message strings output by `perror`. See `perror`(3C) and `intro`(2) for further details.

A/UX Toolbox

The A/UX Toolbox is a set of routines and utilities that make the Macintosh ROM code directly available to a program running under A/UX. It lets you write applications in A/UX that take advantage of the standard Macintosh user interface tools built into the ROMs. For a description of the ROM code, see *Inside Macintosh*.

The A/UX Toolbox bridges the Macintosh and A/UX environments, giving you two kinds of code compatibility:

- You can write common source code that can be separately built (compiled and linked) into executable code for both environments.
- You can execute Macintosh binary files under A/UX, within the limitations of the A/UX Toolbox.

For details on the A/UX Toolbox, please see *A/UX Toolbox: Macintosh ROM Interface*.

Other C language functions

There are numerous other C language functions available under the A/UX operating system designed to handle a variety of tasks. For instance, a very rich set of string functions is available, allowing the programmer to concatenate strings, search for characters within strings, find substrings of strings, determine the length of strings, and so forth. See `string`(3C) for a complete list of the available string functions.

Associated with the string functions are numerous character testing routines. For instance, the function `isascii` returns a nonzero value if its argument is an ASCII character; otherwise it returns zero. There are also several character conversion functions; the function `tolower`, for example, converts its argument to lowercase. For details on these functions, see `ctype`(3C) and `conv`(3C).

The standard C library also contains functions to accomplish time and date manipulation, numeric conversion, group file access, password file access, parameter access, hash table management, random number generation, and so on. A quick browse through Section 3 of *A/UX Programmer's Reference* provides an overview of these various packages.

Other programming tools

In addition to the compilers, language tools, and debuggers already discussed, the A/UX programming environment includes many other useful software development tools. These tools include the following:

`awk`	`awk` is a file-processing language designed to make common information retrieval and manipulation tasks easy to state and to perform. The `awk` language can be used to generate reports, match patterns, validate data, or filter data for transmission.
`bc`	`bc` is a specialized language and compiler for handling arbitrary precision arithmetic using the `dc` calculator program.
`curses`	The `curses` and `terminfo` packages provide a complete set of utility routines for writing screen-oriented programs.
`dc`	`dc` is an interactive desk calculator program for handling arbitrary-precision integer arithmetic. It has provisions for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.
`lex`	`lex` is a lexical analyzer generator that processes character input streams and recognizes regular expressions. It accepts high-level, problem-oriented specifications for character-string matching.
`m4`	`m4` is a general-purpose macro processor. The primary function of `m4` is to allow the replacement of some text by other text. See also the standard C preprocessor (`cpp`).
`make`	The `make` program is a program maintenance tool that keeps track of (and updates) groups of related files. All information about special libraries, special treatments, or options necessary for compiling multiple files is contained in a `make` description file. Using it ensures that all program modules in your compilations reflect your latest changes.
SCCS	The source code control system (SCCS) is a version management tool for source code or text files. In group projects, SCCS prevents multiple inconsistent versions of files from accumulating in several places. For a single user, multiple versions of a file can be stored without using a lot of disk space, previous versions can be reconstructed easily, and versions can be tracked with a simple, consistent numbering scheme.

`yacc`	The `yacc` program is a parser-generator used to impose structure on program input. After you create a specification of the input process, `yacc` generates a parser function, which calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items, called *tokens*, from the input stream. Tokens are organized according to the input structure rules, called *grammar rules*. When one of these rules is recognized, the user code (the *action*) supplied for this rule is invoked. Actions have the ability to return values and make use of the values of other actions.

For information about these tools and how to use them, please refer to *A/UX Programming Languages and Tools,* Volume 2. In addition, the A/UX stream editor `sed` (which operates on a byte-stream rather than an open file) is documented in *A/UX Text Editing Tools,* and all A/UX programs have entries in *A/UX Command Reference, A/UX Programmer's Reference,* or *A/UX System Administrator's Reference.*

As a closing note to this programming overview, you should know that the A/UX shells are themselves fully programmable interpreted languages. Shell scripts, therefore, can sometimes provide very rapid prototyping of programming tasks. As mentioned earlier, it is often a trivial task to translate a shell script into a functionally equivalent C program. So you can begin generating an application program by using the shell tools: pipes, input/output redirection, variables, quotation, and filename substitution. In many instances, these shell scripts can serve as final versions of your program. The shell programming facilities are fully documented in *A/UX Shells and Shell Programming.*

2 `cc` Command Syntax

The `cc` command is a front-end program that invokes the preprocessor, compiler, optimizer, assembler, and link editor, as appropriate. (The default is to invoke each one in turn, except the optimizer, which is invoked only by request.)

This chapter describes the command syntax for `cc` (see also `cc`(1) in *A/UX Command Reference*).

Command syntax

The syntax for `cc` is

`cc [`*flagopt...*`]` *file...*

where *flagopt* is zero or more flag options (see "Options," later in this chapter) and *file* is one or more filenames. `cc` recognizes filenames of the form

file`.`*x*

The two-character extension *.x* identifies the contents of the file, as follows:

Extension	*Contents*	*Example*
`.c`	C source code	`program.c`
`.i`	preprocessor output	`program.i`
`.s`	assembler source	`program.s`
`.o`	assembler output	`program.o`
`.a`	library archive	`libc.a`

A filename with no extension is assumed to be a library archive.

Default behavior

Running `cc` with no flag options on a file named *file*`.c` invokes the C preprocessor, the C compiler, the assembler, and the linkage editor in turn. This process produces an executable file in the current directory; by default, this executable file is named `a.out`.

`cc` has a large number of flag options that can be used to control the compilation process. In addition, other flag options can be passed to the preprocessor, compiler, assembler, and link editor. The sections that follow describe these flag options.

Feature test macros

POSIX specifies certain symbols that are defined in header files. Some of these header files also can define symbols in addition to those defined by POSIX, potentially conflicting with symbols defined by an application program. Feature test macros control the visibility of these symbols in the header files defined by POSIX.

A/UX defines the following feature test macros:

`_AUX_SOURCE`

`_BSD_SOURCE`

`_SYSV_SOURCE`

`_POSIX_SOURCE`

The feature test macros `_SYSV_SOURCE` and `_BSD_SOURCE` represent the historical implementations on which A/UX is based. `_AUX_SOURCE` represents extensions to the historical implementations that are specific to A/UX. `_POSIX_SOURCE` is normally not defined.

The POSIX standard specifies certain symbols that are defined in header files. Some of these header files can also define symbols in addition to those defined by POSIX, potentially conflicting with symbols defined by an application program. Feature test macros control the visibility of these symbols. When POSIX compilation is selected (with the `-ZP` option), test macros other than `_POSIX_SOURCE` are not defined.

Options

All options recognized by the `cc` command are listed on the following pages.

Recognized and executed by `cc`

Option	*Argument*	*Description*
`-A`	*factor*	Expand the default symbol table allocations for the compiler, assembler, and link editor. The default allocation is multiplied by the factor given.
`-a`	none	Include source code as comments in the assembly file generated with the `-S` option.
`-c`	none	Suppress the link-editing phase of compilation and force a relocatable object file to be produced even if only one file is compiled.
`-F`	none	Do not generate inline code for MC68881 floating-point coprocessor.
`-f`	`m68881`	Generate inline code for MC68881 floating-point coprocessor. This is the default.
`-g`	none	Produce symbolic debugging information.
`-n`	none	Arrange for the loader to produce an executable file that is linked in such a manner that the text can be made read-only and shared (nonvirtual) or pages (virtual).
`-p`	none	Add code to support profiling of the program (see `profil`(1).
`-S`	none	Compile the named C programs, and leave the assembler-language output within corresponding files with `.s` suffixes.

`-t`	`[p012al]`	Find only the designated preprocessor (`p`), compiler (`0`), optimizer (`2`), assembler (`a`), and link editor (`l`) passes whose names are constructed with the *string* argument to the `-B` option. In the absence of a `-B` option and its argument, *string* is taken to be `/lib/n`.
`-B`	*string*	Construct pathnames for substitute preprocessor, compiler, and link editor passes by concatenating *string* with the suffixes `cpp`, `ccom`, `optim`, `as`, and `ld`. The passes affected may be specified by the `-t` option. In the absence of the `-t` option, all passes are affected.
`-E`	none	Same as the `-P` option except output is directed to the standard output.
`-O`	none	Invoke a code optimizer.
`-o`	*outfile*	Specify the name of the final object file, *outfile*. This object file is the output of the loader unless the `-c` option is specified, in which case it is the ouput of the assembler. The default ouput file for the loader is `a.out`. The default for the assembler is `file.o`.
`-P`	none	Suppress compilation and loading; that is, invoke only the preprocessor and leave the output on corresponding files with the `.i` extension.
`-R`	none	Have assembler remove its input file when done.
`-T`	none	Truncate symbol names to eight significant characters.
`-v`	none	Print the command line for each subprocess executed.
`-W`	*c,arg1* [,*arg2*...]	Pass the argument(s) *arg1* to *c*, where *c* is one of `[p0112al]`, indicating preprocessor (`p`), compiler first pass (`0`), compiler second pass (`1`), optimizer (`2`), assembler (`a`), or link editor (`l`), respectively.
`-X`	none	Ignored by A/UX for 68000-family processors.

Option	*Argument*	*Description*
-Z	*flags*	Special *flags* to override the default behavior (see cc(1)). Currently recognized *flags* are as follows: c Return pointers in a0 without copying to d0. n Emit no code for stack growth. m Use Motorola SGS compatible stack growth code. p Use tst.b stack probes. E Ignore all environment variables. I Emit instructions for MC68881 floating-point coprocessor. l Suppress selection of a loader command file. t Do not delete temporary files. P Compile for the A/UX POSIX environment. Link the file with a library module that calls setcompat(2) with the COMPAT_POSIX flag set. Define only the _POSIX_SOURCE feature test macro. See Appendixes B and C for more information on the POSIX environment and conformance requirements. S Compile to be SVID-compatible. Link the file with a library module that calls setcompat(2) with the COMPAT_SVID flag set. Define only the _SVSV_SOURCE feature test macro. B Compile to be BSD-compatible. Link the file with a library module that calls setcompat(2) with the COMPAT_BSD flag set. Define only the _BSD_SOURCE feature test macro.
-#	none	Special debug option that, without actually starting the program, echoes the names and arguments of subprocesses that would have started.

Recognized by `cc` and passed to `as`

The following options are recognized by `cc` and passed to the assembler. They are only needed if you are running the assembler on hand-written files that contain instructions not available with the Motorola 68020 processor. The compiler does not generate such instructions.

Option	*Argument*	*Description*
`-68851`	none	Directs the assembler to recognize the coprocessor instructions for a Motorola 68851 PMMU. This is the default.
`-68030`	none	Directs the assembler to recognize the memory management unit (MMU) instructions for a Motorola 68030 processor.
`-68040`	none	Directs the assembler to recognize the instructions for a Motorola 68040 processor.

Recognized by `cc` and passed to `ld`

Option	*Argument*	*Description*
`-l`	*name*	Same as `-l` in `ld`(1). Search a library `lib`*x*`.a`, where *x* is up to seven characters. A library is searched when its name is encountered, so the placement of a `-l` is significant. By default, libraries are located in `LIBDIR`. If you plan to use the `-L` option, that option must precede `-l` on the command line.
`-s`	none	Same as `-s` in `ld`(1). Strip line number entries and symbol table information from the output of object file.
`-L`	*dir*	Same as `-L` in `ld`(1). Search for `lib`*name*`.a` in the named *dir* before looking in `LIBDIR`. This option is effective only if it precedes the `-l` option on the command line.
`-V`	none	Print the version of the loader that is invoked.

Recognized by `cc` and passed to `cpp`

Option	*Argument*	*Description*
`-C`	none	Same as `-C` in `cpp`(1). All comments, except those found on `cpp` directive lines, are passed along. The default strips out all comments.
`-D`	*symbol*[=*def*]	Same as `-D` in `cpp`(1). Define the external *symbol* and give it the value *def* (if specified). If no *def* is given, *symbol* is defined as 1.
`-I`	*dir*	Search for `#include` files that do not begin with / in the named *dir* before looking in the directories on the standard list. Thus, `#include` files whose names are enclosed in `""` (for example, `#include "thisfile"`) are first searched for in the directory of the file being compiled, then in directories named by the `-I` *options*, and last in directories on the standard list. For `#include` files whose names are enclosed in `<>` (for example, `#include <thisfile>`), the directory of the file being compiled is not searched.
`-U`	*symbol*	Remove any initial definition of *symbol* ("undefine" *symbol*), where *symbol* is a reserved name that is predefined by the particular preprocessor.
`-y`	none	Suppress searching of `/usr/include` for header files and instead search only in directories specified by the `-I` option.

By using appropriate options, you can terminate compilation early to produce one of several intermediate translations, as follows:

`-c`	This option produces relocatable object files. It is often desirable to use this option to save relocatable files so that changes to one file do not then require that the other files be recompiled. A separate call to `cc`, with the relocatable files but without the `-c` option, creates the linked executable `a.out` file. A relocatable object file created under the `-c` option has the same basename as the relocatable object file, but the extension is `.o` instead of `.c`.
`-S`	This option produces assembly source expansions for C code.
`-P`	This option produces the output of the preprocessor. When you use this option, the compilation process stops after preprocessing. Output from the preprocessor is left in an output file with the extension `.i` (for example, `file1.i`). These output files can be subsequently processed by `cc`, but only if their filename is changed to one with the extension `.c`. Except for those produced by the preprocessor, any intermediate files can be saved and resubmitted to the `cc` command, with other files or libraries included as necessary.

For more information on any of the options that `cc`(1) passes to the preprocessor `cpp`(1), the link editor `ld`(1), or the assembler `as`(1), see the appropriate manual page in *A/UX Command Reference*.

3 C Language Reference

This chapter describes the C programming language. The manner of presentation of C syntax is meant to help you gain understanding of the language structure. It should not be taken as a formal definition of the language.

Notation conventions

In the syntax notation used in this chapter, syntactic categories are indicated by *italic* type and literal words and characters by `courier` type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript *opt*, so that

[$expression_{opt}$]

indicates an optional expression enclosed in braces. The syntax is summarized in "Syntax Summary," the final section in this chapter.

Lexical conventions

There are six classes of tokens:

1. identifiers
2. keywords
3. constants
4. strings
5. operators
6. other separators

Blanks, tabs, newlines, and comments (collectively called *white space*) are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream is parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

```
/* Comments/* do not*/ nest*/
```

◆ **Note** The above comment terminates after the `*/` following `not`, leaving `nest*/` to be read as code. ◆

As an extension to the C language, `cc` recognizes `//` to mean the rest of the line is a comment. For example,

```
i=10;        //initialize counter ...comment to end of line
```

Identifiers (names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are read differently and are not interchangeable. Although there is no length limit for names, only the initial 256 characters of the name are significant. This implementation accepts identifiers up to 1024 characters long. Other implementations truncate identifiers to 7 or 8 characters, so long identifier names are not recommended.

Keywords

The following identifiers are reserved for use as keywords and cannot be used otherwise:

```
asm          do           fortran      short        unsigned
auto         double       goto         sizeof       void
break        else         if           static       while
case         enum         int          struct
char         extern       long         switch
continue     float        register     typedef
default      for          return       union
```

Constants

There are several kinds of constants, each of which has a type. The introduction to types is given in the "Names" section, and hardware characteristics that affect sizes are summarized in the subsection "Hardware Characteristics," both later in this chapter. See also Chapter 4, "C Implementation Notes."

Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with a zero. An octal constant consists of the digits 0 through 7 only. A sequence of digits preceded by `ox` or `0x` is taken to be a hexadecimal integer. The hexadecimal digits include *a* through *f* (or *A* through *F*) with corresponding decimal values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be `long`. An octal or hexadecimal constant that exceeds the largest unsigned machine integer is likewise taken to be `long`. Otherwise, integer constants are `int`.

Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by the letter `l` or `L` is a `long` constant. As discussed later, on the Macintosh 68000 family of computers `integer` and `long` values are considered identical.

Character constants

A character constant is a character enclosed in single quotes, as in `'x'`. The value of a character constant is the numeric value of the character in the machine character set.

Multicharacter character constants are permitted on the 68000-family processors. Multicharacter character constants can be differentiated from strings by the following criterion: strings are enclosed in double quotes (`" "`), while multicharacter character constants are enclosed in single quotes (`' '`). Characters are assigned to a word left to right. For example, when you compile a program including the line

```
i = 'abcd';
```

`i` is assigned the value `0x61626364`.

Two metacharacters, the single quote (`'`) and the backslash (`\`), are used in escape sequences. To use these characters literally, they must be "escaped" as shown in Table 3-1.

Table 3-1 Character constants and escape sequences

Character	ASCII	Escape sequence
Null	NUL	`\0`
Newline	NL(LF)	`\n`
Horizontal tab	HT	`\t`
Vertical tab	VT	`\v`
Backspace	BS	`\b`
Carriage return	CR	`\r`
Form feed	FF	`\f`
Backslash	`\`	`\\`
Single quote	`'`	`\'`
Bit pattern	`^\`*onum*	`\`*onum*

The escape `\`*onum* consists of the backslash followed by 1, 2, or 3 octal digits (0 through 7), which are taken to specify the value of the desired character. If the character following a backslash is not one of those specified, the behavior is undefined. A newline character is illegal in a character constant. The type of a character constant is `int`.

Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part can be missing, but not both. Either the decimal point or the `e` and exponent can be missing, but not both. Every floating constant has type `double`.

Enumeration constants

Names declared as enumerators have type `int`. For more information, see the sections "Structure and Union Declarations" and "Enumeration Declarations," later in this chapter.

Strings

A string is a sequence of characters surrounded by double quotes, as in `"string"`. A string has type *array of* `char` and storage class `static` and is initialized with the given characters. The compiler places a `null` byte (`\0`) at the end of each string so that programs scanning the string can find its end. In a string, the double-quote character (`"`) must be preceded by a backslash (`\`). In addition, the same escapes as described for character constants can be used.

A backslash (`\`) and the newline immediately following it are ignored. All strings, even when formally identical, are distinct.

As an extension, a string may be prefixed by `\p` to indicate that it is a Pascal string. It is stored internally with both the terminating null byte and a preceding length count. For example, `\pabc` is stored as `0x03616200`.

Hardware characteristics

Table 3-2 summarizes certain hardware properties for the 68000-family processors. Note that the ranges for `float` and `double` are approximate.

For more information on 68000-family data representation, see Chapter 4, "C Implementation Notes."

Table 3-2 68000-family hardware characteristics

Type	**Representation**
`char`	8 bits
`short`	16
`int`	32
`long`	32
`float`	32
`double`	64
`float` *range*	$\pm 10^{\pm 38}$
`double` *range*	$\pm 10^{\pm 307}$

Names

The C language bases the interpretation of an identifier upon two attributes of the identifier:

- **Storage class** determines the location and lifetime of the storage associated with an identifier.
- **Type** determines the meaning of the values found in the storage of the identifier.

Storage class

There are four declarable storage classes:

- **Automatic variables** are local to each invocation of a block and are discarded after exiting the block.
- **Static variables** are local to a block, but retain their values upon reentry to a block even after control leaves the block.
- **External variables** exist and retain their values throughout the execution of the entire program. They can be used for communication among functions, even separately compiled functions.
- **Register variables** are stored in the fast registers of the machine until these registers run out. The remainder are treated as automatic variables. Like automatic variables, they are local to each block and disappear after exiting the block.

Type

The C language supports several fundamental types of objects. Objects declared as characters (`char`) are large enough to store any member of the implementation character set. If a genuine character from that character set is stored in a `char` variable, its value is equivalent to the integer code for that character. Other quantities can be stored in character variables, but the implementation is machine dependent. In particular, `char` can be signed or unsigned, by default.

Up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Longer integers do not provide less storage than shorter ones, but the implementation can make short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs. (See "Hardware Characteristics," earlier in this chapter, for the sizes of types on the 68000-family processor.)

`enum` types have the same size as an `int` or `long`. The properties of `enum` types are identical to those of some integer types, with the exceptions that some conversions to or from them are not allowed (for example, with `float`) and that they can be compared only for equality.

Unsigned integers, declared `unsigned`, obey the laws of arithmetic modulo 2^n, where n is the number of bits in the representation.

Because objects of these types can be usefully interpreted as numbers, they are referred to as **arithmetic types.** The `char` and `int` types of all sizes, whether `unsigned` or not, and the `enum` type are collectively called **integral types.** The `float` and `double` types are collectively called **floating types.** Table 3-3 summarizes the categorization of fundamental types.

Table 3-3 Categorization of fundamental types

	Category		
Type	**Arithmetic**	**Integral**	**Floating**
`char`	x	x	
`double`	x		x
`enum`		x	
`float`	x		x
`int`	x	x	
`long`	x	x	
`short`	x	x	

In addition to the fundamental arithmetic types, there is a conceptually infinite class of derived types, constructed from the fundamental types in the following ways:

- arrays of objects of most types
- functions that return objects of a given type
- pointers to objects of a given type
- structures containing a sequence of objects of various types
- unions capable of containing any one of several objects of various types

In general, these methods of constructing objects can be applied recursively.

Objects and lvalues

An **object** is a region of storage that can be manipulated. An **lvalue** is an expression referring to an object—for example, an identifier. There are operators that yield lvalues. For example, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name "lvalue" comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator that follows indicates whether it expects lvalue operands and whether it yields an lvalue.

Conversions

A number of operators can, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result you can expect from such conversions. The conversions demanded by most ordinary operators are summarized later in this chapter in "Arithmetic Conversions."

Characters and integers

A `char` or a `short` can be used wherever an `int` is allowed. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer one preserves sign. Whether sign extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. A character constant specified with an octal escape, however, suffers sign extension and can appear negative; for example, `'\377'` has the value `-1`.

When a longer integer is converted to a shorter integer or to a `char`, it is truncated on the left. Excess bits are simply discarded.

Float and double

All floating arithmetic in C is carried out in double precision. Whenever a `float` appears in an expression, it is lengthened to `double` by right-padding its fraction with zeros. When a `double` must be converted to `float`—for example, by an assignment—the `double` is rounded before truncation to `float` length. This result is undefined if it cannot be represented as a `float`.

Floating and integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. On the 68000 family, negative floating values are rounded toward zero. The result is undefined if it does not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

Pointers and integers

An expression of integral type can be added to or subtracted from a pointer (thus, **pointer arithmetic** is allowed). In such a case, the first is converted as specified in the discussion of the addition operator (later). Two pointers to objects of the same type can be subtracted. In this case, the result is converted to an integer, as specified in the discussion of the subtraction operator (later).

Unsigned

Whenever an unsigned integer and a signed integer are combined, the signed integer is converted to unsigned and the result is unsigned. In a 2's-complement representation, this conversion is conceptual, and there is no actual change in the bit pattern. The value of the converted integer is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$).

When an unsigned short integer is converted to long, the value of the result is the same numerically as that of the unsigned integer. Thus, the conversion amounts to padding with zeros on the left.

Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. From here on in this document, this pattern is called the *usual arithmetic conversions.* These rules are applied in the order in which they appear, if applicable.

◆ **Note** In this implementation, `int` and `long` have the same size and do not require conversions to or from each other. In the following table, therefore, `long` is used in place of `int`. ◆

Conversions are performed only if necessary, depending on the operation. If a `char` is added to a `char`, the result remains a `char`. If an `int` is the result of adding a `char` to a `char`, the conversion is done before the addition.

- First, `char` or `short` is converted to `long`, and `unsigned char` or `unsigned short` is converted to `unsigned long`. `float` is converted to `double`.
- Next, if either operand is `double`, the other one converts to `double` and the result is `double`.
- Next, if either operand is `unsigned long`, the other one converts to `unsigned long` and the result is `unsigned long`.
- Next, if either operand is `long`, the other one converts to `long` and the result is `long`.
- Next, if either operand is `unsigned`, the other one converts to `unsigned` and the result is `unsigned`.
- Finally, if both operands are `long`, the result is `long`.

Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. For example, the expressions referred to as the operands of + are those expressions defined in "Primary Expressions," "Unary Operators," and "Multiplicative Operators," later in this chapter. Within each subpart, the operators have the same precedence. Left or right associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar in "Syntax Summary," near the end of this chapter.

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (`*, +, &, |, ^`) can be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation, your program must use an explicit temporary location.

The behavior of overflow and divide by zero exceptions in expression evaluation is undefined. This implementation, like most that exist, ignores integer overflows. The integer division by zero exception is enabled by default. The result of an integer division by zero can be detected using `adb` on the assembler file—it is designated `Inf` (infinity) or `NaN` (not a number). All other floating-point exceptions are disabled. For more information on the floating-point exception, see the Motorola *MC68881 Floating Point Coprocessor User's Manual,* Motorola part number M68KMASM.

Primary expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

primary-expression:

> *identifier*
> *constant*
> *string*
> `(` *expression* `)`
> *primary-expression* `(` *expression* `)`
> *primary-expression* `(` *expression-list*$_{opt}$ `)`
> *primary-expression* `.` *identifier*
> *primary-expression* `->` *identifier*

expression-list:

> *expression*
> *expression-list* `,` *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed later. Its declaration specifies its type. If the identifier type is

array of some-type

the value of the identifier expression is a pointer to the first object in the array, and the type of the expression is

pointer to some-type

Moreover, an array identifier is not an lvalue expression. Likewise, an identifier that is declared

function returning some-type

when used, except in the function-name position of a call, is converted to

pointer to function returning some-type

A constant is a primary expression. Its type can be `int`, `long`, or `double`, depending on its form. Character constants have type `int` and floating constants have type `double`.

A string is a primary expression. Its type is originally *array of* `char`, but following the same rule given earlier for identifiers, this is modified to *pointer to* `char`. The result is a pointer to the first character in the string (there is an exception in certain initializers; see "Initialization," later in this chapter).

A parenthetical expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type

pointer to some-type

The subscript expression is `int`, and the type of the result is

some-type

The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. All the clues needed to understand this notation are contained in this subsection together with the discussions in "Unary Operators" and "Additive Operators" on identifiers * and +, respectively. The implications are summarized in "Arrays, Pointers, and Subscripting" under "Types Revisited," later in this chapter.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions that constitute the actual arguments to the function. The primary expression must be of type

function returning some-type

and the result of the function call is of type

some-type

As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer. Therefore, in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call. Any arguments of type `char` or `short` are converted to `int`. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast. For further information, see "Unary Operators" and "Type Names," later in this chapter.

In preparing for the call to a function, a copy is made of each actual parameter. Thus, *all argument passing in C is strictly by value.* A function can change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function can change the value of the object to which the pointer points. An array name is a pointer expression; therefore, in effect, array arguments are passed by reference. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot, followed by an identifier, is an expression. The primary expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is that named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from `-` and `>`), followed by an identifier, is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue that refers to the named member of the structure or union to which the pointer expression points. Thus, the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in greater detail in "Structure and Union Declarations" and "Enumeration Declarations" under "Declarations."

Unary operators

Expressions with unary operators group right to left.

unary-expression:

`*` *expression*
`&` *lvalue*
`-` *expression*
`!` *expression*
`~` *expression*
`++` *lvalue*
`--` *lvalue*
lvalue `++`
lvalue `--`
`(` *type-name* `)` *expression*
`sizeof` *expression*
`sizeof (`*type-name*`)`

The unary operator `*` means *indirection;* the expression must be a pointer and the result is an lvalue referring to the object to which the expression points. If the type of the expression is

pointer to some-type

the type of the result is

some-type

The result of the unary `&` operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is

some-type

the type of the result is

pointer to some-type

The result of the unary `-` operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in the corresponding signed type.

There is no unary `+` operator.

The result of the logical negation operator `!` is one (1) if the value of its operand is zero, and zero if the value of its operand is nonzero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the 1's-complement of its operand. The usual arithmetic conversions are performed. The operand must be of the integral type.

The object referred to by the lvalue operand of prefix `++` is incremented. The value is the new value of the operand but is not an lvalue. The expression `++x` is equivalent to `x = x + 1`. See "Additive Operators" and "Assignment Operators," later in this chapter, for information on conversions.

The lvalue operand of prefix `--` is decremented analogously to the prefix `++` operator.

When postfix `++` is applied to an lvalue, the result is the value of the object to which the lvalue refers. After the result is noted, the object is incremented in the way the prefix `++` operator was implemented. The type of the result is the same as the type of the lvalue expression.

When postfix `--` is applied to an lvalue, the result is the value of the object to which the lvalue refers. After the result is noted, the object is decremented in the same manner as the prefix `--` operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes the expression value to convert to the named type. This construction is called a **cast.** Type names are described in "Type Names," later in this chapter.

The `sizeof` operator yields its operand size in bytes. (A **byte** is undefined by the language except in terms of the value of `sizeof`. In this implementation, as in all existing ones, however, a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an `unsigned` constant and can be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator also can be applied to a type name enclosed in parentheses. In that case it yields the size, in bytes, of an object of the indicated type.

The construction `sizeof(`*type*`)` is taken to be a unit, so the expression `sizeof(`*type*`)-2` is the same as `(sizeof(`*type*`))-2`.

Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left to right. The usual arithmetic conversions are performed.

multiplicative expression:

> *expression* `*` *expression*
> *expression* `/` *expression*
> *expression* `%` *expression*

The binary `*` operator indicates multiplication. The `*` operator is associative and expressions with several multiplications at the same level can be rearranged by the compiler. The binary `/` operator indicates division.

The binary `%` operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward zero. The remainder has the same sign as the dividend. It is always true that `(a/b)*b + a%b` is equal to `a` (if `b` is not zero).

Additive operators

The additive operators `+` and `-` group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:

> *expression* `+` *expression*
> *expression* `-` *expression*

The result of the `+` operator is the sum of the operands. A pointer to an object in an array and a value of any integral type can be added. The latter is, in all cases, converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, which points to another object in the same array, appropriately offset from the original object. Thus, if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The `+` operator is associative, and expressions with several additions at the same level can be rearranged by the compiler.

The result of the `-` operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type can be subtracted from a pointer and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (through division by the length of the object) to an `int` representing the number of objects pointed to. This conversion, in general, gives unexpected results unless the pointers point to objects in the same array; pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

Shift operators

The shift operators `<<` and `>>` group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

shift-expression:

expression `<<` *expression*
expression `>>` *expression*

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits. Vacated bits are zero filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (zero fill) if `E1` is `unsigned`; otherwise, it can be arithmetic.

Relational operators

The relational operators group left to right.

relational-expression:

expression `<` *expression*
expression `>` *expression*
expression `<=` *expression*
expression `>=` *expression*

The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield zero if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed. You can compare two pointers; the result depends on the relative locations in the address space of the objects pointed to. Pointer comparison is portable only when the pointers point to objects in the same array.

Equality operators

equality-expression:

expression `==` *expression*
expression `!=` *expression*

The `==` (equal to) and the `!=` (not equal to) operators are exactly analogous to the relational operators, except they have lower precedence (thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth value).

You can compare a pointer to an integer only if the integer is the constant zero. A pointer to which zero has been assigned is guaranteed not to point to any object and appears to be equal to zero. In conventional usage, such a pointer is considered to be *null*.

Bitwise AND operator

and-expression:

expression `&` *expression*

The `&` operator is associative; expressions involving `&` can be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

Bitwise exclusive OR operator

exclusive-or-expression:

expression `^` *expression*

The `^` operator is associative; expressions involving `^` can be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

Bitwise inclusive OR operator

inclusive-or-expression:

expression `|` *expression*

The `|` operator is associative; expressions involving `|` can be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

Logical AND operator

logical-and-expression:

expression `&&` *expression*

The `&&` operator groups left to right. It returns 1 if both its operands evaluate to nonzero; otherwise it returns zero. Unlike `&`, `&&` guarantees left-to-right evaluation. Moreover, the second operand is not evaluated if the first operand is zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

Logical OR operator

logical-or-expression:

expression `||` *expression*

The `||` operator groups left to right. It returns 1 if either of its operands evaluates to nonzero; otherwise it returns zero. Unlike `|`, `||` guarantees left-to-right evaluation. Moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

Conditional operator

conditional-expression:

expression `?` *expression* `:` *expression*

Conditional expressions group right to left. The first expression is evaluated. If it is nonzero, the result is the value of the second expression; otherwise, that of the third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has that type as well. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant zero, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

Assignment operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand. The type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment takes place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:

lvalue `=` *expression*
lvalue `+=` *expression*
lvalue `-=` *expression*
lvalue `*=` *expression*
lvalue `/=` *expression*
lvalue `%=` *expression*
lvalue `>>=` *expression*
lvalue `<<=` *expression*
lvalue `&=` *expression*
lvalue `^=` *expression*
lvalue `|=` *expression*

In the simple assignment with `=`, the value of the expression replaces that of the object to which the lvalue refers. If both operands have arithmetic type, the right operand is converted to the type of the left, preparatory to the assignment. If both operands are structures or unions, they must be of the same type. If the left operand is a pointer, the right operand must, in general, be a pointer of the same type. The constant zero can be assigned to a pointer, however—it is guaranteed that this value produces a null pointer that is distinguishable from a pointer to any object.

You can understand the behavior of an expression of the form `E1` *op* `= E2` by taking it as equivalent to `E1 = E1` *op*`(E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand can be a pointer, in which case the (integral) right operand is converted as explained in "Additive Operators," earlier in this chapter. All right operands and all nonpointer left operands must have arithmetic type.

Comma operator

comma-expression:

  expression, *expression*

A pair of expressions separated by a comma is evaluated left to right. The value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. It is useful in situations where you wish to combine operations on one line and do not care about seeing the first result, just about using it in the second operation. In contexts where a comma is given a special meaning, for example, in lists of actual arguments to functions (see "Primary Expressions," earlier in this chapter) and lists of initializers (see "Initialization," later in this chapter), the comma operator, as described in this section, can appear only in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

Declarations

Declarations are used to specify the interpretation that C gives to each identifier. They don't necessarily reserve storage associated with the identifier. Declarations have the form

declaration:

  decl-specifiers declarator-list$_{opt}$;

The declarators in the *declarator-list* contain the identifiers being declared. The *decl-specifiers* consist of a sequence of type and storage class specifiers.

decl-specifiers:

  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent, as the following section describes.

Storage class specifiers

The storage class specifiers are as follows:

```
auto
static
extern
register
typedef
```

The `typedef` specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience (see "Typedef," later in this chapter). The meanings of the various storage classes are discussed in "Names," earlier in this chapter.

The `auto`, `static`, and `register` declarations also serve as definitions because they cause an appropriate amount of storage to be reserved. In the `extern` case, there must be an external definition (see "External Definitions," later in this chapter) for the given identifiers, somewhere outside the function in which they are declared.

A `register` declaration is best thought of as an `auto` declaration that hints to the compiler that the variables declared are to be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types are stored in registers. One other restriction applies to register variables: The address-of operator `&` cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately.

At most, one storage class specifier can be given in a declaration. If the storage class specifier is missing from a declaration, it is taken to be `auto` inside a function, `extern` outside.

◆ **Note** The exception is that functions are never automatic. ◆

Type specifiers

The type specifiers are as follows:

type-specifier:

struct-or-union-specifier
basic-type-specifier
typedef-name
enum-specifier

basic-type-specifier:

basic-type
basic-type basic-type-specifier

basic-type:

```
char
short
int
long
unsigned
float
double
```

`long` or `short` can be specified in conjunction with `int`; the meaning is the same as if `int` were not mentioned. The word `long` can be specified in conjunction with `float`; the meaning is the same as `double`. `unsigned` can be specified alone or in conjunction with `int` or any of its `short` or `long` varieties or with `char`.

Except for the combinations just described, only a single type specifier can be given in a declaration. In particular, using `long`, `short`, or `unsigned` as an adjective is not permitted with `typedef` names. If the type specifier is missing from a declaration, it is taken to be `int`.

Specifiers for structures, unions, and enumerations are discussed later in this chapter in "Structure and Union Declarations" and "Enumeration Declarations." Declarations with `typedef` names are discussed in "Typedef," also later in this chapter.

Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer, as follows:

declarator-list:

init-declarator

init-declarator, *declarator-list*$_{opt}$

init-declarator:

declarator initializer$_{opt}$

Initializers are discussed in "Initialization," later in this chapter. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax

declarator:

identifier

(*declarator*)

* *declarator*

declarator ( )

declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier: this is what is being declared. If an unadorned identifier appears as a declarator, it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators can be altered by parentheses (see the following examples).

Now imagine a declaration:

T D1

where *T* is a type specifier (for example, `int`) and *D1* is a declarator. Suppose this declaration declares the identifier to be of type

[*modifier*] *T*

where the [*modifier*] is empty if *D1* is just a plain identifier (so that the type of *x* in `int` *x* is just `int`). Then if *D1* has the form

**D*

the type of the contained identifier is

[*modifier*] *pointer to T*

If *D1* has the form

D()

the contained identifier has the type

[*modifier*] *function returning T*

If *D1* has the form

D[*constant-expression*]

or

D[]

the contained identifier has type

[*modifier*] *array of T*

In the first case, the constant expression is an expression whose value can be determined at compile time, whose type is `int`, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions," later in this chapter.) When several *array of* specifications are adjacent, a multidimensional array is created. The constant expressions that specify the bounds of the arrays can be missing only for the first member of the sequence. This missing section is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression also can be omitted when the declarator is followed by initialization. In this case, the size is calculated from the number of initial elements supplied.

An array can be constructed from one of the basic types, from a pointer, a structure or union, or from another array (to generate a multidimensional array).

Not all possibilities of the above syntax are actually permitted. The restrictions are as follows: Functions cannot return arrays or functions although they can return pointers; there are no arrays of functions, although there can be arrays of pointers to functions; likewise, a structure or union cannot contain a function, but it can contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares

`i`	an integer
`*ip`	a pointer to an integer
`f()`	a function returning an integer
`*fip()`	a function returning a pointer to an integer
`(*pfi)()`	a pointer to a function that returns an integer

It is especially useful to compare the last two:

`*fip()`	The binding of `*fip()` is `*(fip())`. If this declaration were part of an expression in the code, it would call the function `fip`. `fip` returns a pointer. Using indirection through this pointer yields an integer.
`(*pfi)()`	In the declarator `(*pfi)()`, or such a construct in an expression, the parentheses must enclose `*pfi` to show that the whole thing yields a function (through indirection through a pointer). When this function is called, it returns an integer.

As an example,

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers.

Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3x5x7. In complete detail, `x3d` is an array of three items. Each item is an array of five arrays. Each of the arrays is an array of seven integers.

Any of the expressions

```
x3d
x3d[i]
x3d[i][j]
x3d[i][j][k]
```

can reasonably appear in an expression. The first three have type *array* and the last has type `int`.

Structure and union declarations

A structure is an object made up of a sequence of named members. Each member can have any type. A union is an object that can, at a given time, contain any one of several members. Structure and union specifiers have the same form:

struct-or-union-specifier:

> *struct-or-union* { *struct-decl-list* }
> *struct-or-union identifier* { *struct-decl-list* }
> *struct-or-union identifier*

struct-or-union:

> `struct`
> `union`

The *struct-decl-list* is a sequence of declarations for the members of the structure or union:

struct-decl-list:

> *struct-declaration*
> *struct-declaration struct-decl-list*

struct-declaration:

> *type-specifier struct-declarator-list;*

struct-declarator-list:

> *struct-declarator*
> *struct-declarator, struct-declarator-list*

In the usual case, a *struct-declarator* is just a declarator for a member of a structure or union. A structure member also can consist of a specified number of bits. Such a member is also called a *field;* its length, a non-negative constant expression, is set off from the field name by a colon.

struct-declarator:

declarator
declarator : *constant-expression*
: *constant-expression*

Within a structure, the objects declared have addresses that increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there can be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field that does not fit into the space remaining in a word is put into the next word. No field can be wider than a word.

A *struct-declarator* with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally imposed layouts. As a special case, a field with a width of zero specifies alignment of the next field on an implementation-dependent boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any except integer fields. Moreover, even `int` fields can be considered to be unsigned.

It is strongly recommended that you declare fields as unsigned. In all implementations, there are no arrays of fields, and the address-of operator `&` cannot be applied to them, so that there are no pointers to fields.

A union can be thought of as a structure, all of whose members begin at offset zero and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form,

`struct` *identifier* { *struct-decl-list* }
`union` *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration can then use the third form of specifier,

`struct` *identifier*
`union` *identifier*

Structure tags allow definition of self-referencing structures. They also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union that contains an instance of the structure or union itself, but it can contain a pointer to an instance of itself.

You can use the third form of a structure or union specifier before a declaration that gives the complete specification of the specifier in situations in which its size is unnecessary. The size is unnecessary in two situations: (1) when a pointer to a structure or union is being declared and (2) when a `typedef` name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures that contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name cannot be used twice in the same structure, but the same name can be used in several different structures in the same scope.

A simple but important example of a structure declaration is the binary tree structure,

```
struct tnode
{
            char tword[20];
            int count;
            struct tnode *left;
            struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration is given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be a structure of the given sort and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the `count` field of the structure to which `sp` points;

```
s.left
```

refers to the left subtree pointer of the structure `s`; and

```
s.right->tword[0]
```

refers to the first character of the `tword` member of the right subtree of `s`.

Enumeration declarations

Enumeration variables and constants have *integral* type.

enum-specifier:

    `enum {`*enum-list*`}`
    `enum` *identifier* `{`*enum-list*`}`
    `enum` *identifier*

enum-list:

    enumerator
    enum-list , *enumerator*

enumerator:

    identifier
    identifier = *constant-expression*

The identifiers in an *enum-list* are declared as constants and can appear wherever constants are required. If no enumerators with = appear, the values of the corresponding constants begin at zero and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the *enum-specifier* is entirely analogous to that of the structure tag in a *struct-specifier*; it names a particular enumeration. For example,

```
enum color {mauve,burgundy,claret=20,wine} ;
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes `color` the *enumeration-tag* of a type describing various colors and then declares `cp` as a pointer to an object of that type and `col` as an object of that type. The possible values are drawn from the set `{0, 1, 20, 21}`.

Initialization

A declarator can specify an initial value for the identifier being declared. The initializer is preceded by `=` and consists of an expression or a list of values nested in braces.

initializer:

```
        = expression
        = { initializer-list}
        = { initializer-list, }
```

initializer-list:

```
        expression
        initializer-list , initializer-list
        { initializer-list}
        initializer-list, }
```

All the expressions in an initializer for a static or external variable must be constant expressions (see "Constant Expressions," later in this chapter) or expressions that reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables can be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are undefined.

When an initializer applies to a **scalar** (a pointer or object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; it is converted in the same way it would be in an assignment.

When the declared variable is an **aggregate** (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, the aggregate is padded with zeros. You cannot initialize unions or automatic aggregates.

You can, in some cases, omit braces. If the initializer begins with a left brace, the succeeding comma-separated list of initializers initializes the members of the aggregate; the compiler reports an error if there are more initializers than members. If, however, the initializer does not begin with a left brace, only sufficient elements to account for the members of the aggregate are taken from the list; any remaining members are left to initialize the next aggregate member.

A final abbreviation allows a `char` array to be initialized by a string. In this case, successive characters of the string initialize the members of the array.

The syntax of `char` array initialization can be derived from that of numerical array initialization. For example, the construct

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array that has three members, as no size was specified and there are three initializers.

Now consider an example of two-dimensional array initialization. The construct

```
float y[4][3] =
{
            {1, 3, 5},
            {2, 4, 6},
            {3, 5, 7},
};
```

gives a completely bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely,

```
y[0] [0]
y[0] [1]
y[0] [2]
```

Likewise, the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with zero. Precisely the same effect can be achieved with

```
float y[4][3] =
{
            1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace but the one for `y[0]` does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for `y[1]` and `y[2]`.

Also,

```
float y[4][3] =
{
            { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest zero.

A further leap allows for the syntax of character array initialization. Because commas are common elements within strings, it is handier not to have to separate elements with them. It is preferable in this situation to presuppose a variable-length one-dimensional array, the successive elements of which become array members. The array ends when the string is exhausted, as in the two-dimensional array example, and no commas are needed, as the initialization happens all at once. Thus, the construct

```
static char msg[ ] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string. Note the lack of size specification, as in the one-dimensional array example.

Type names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of `sizeof`), you should supply the name of a data type. Your program can do this by using a **type name,** which, in essence, is a declaration for an object of the type that omits the name of the object.

type-name:

    type-specifier abstract-declarator

abstract-declarator:

    empty

    (*abstract-declarator*)

    * *abstract-declarator*

    abstract-declarator ( )

    abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(*abstract-declarator*)

the *abstract-declarator* is required to be nonempty. Under this restriction, your program can identify uniquely the location in the *abstract-declarator* where the identifier appears if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

`int`	is type integer
`int *`	is type pointer to integer
`int *[3]`	is type array of three pointers to integers
`int (*)[3]`	is type pointer to an array of three integers
`int *()`	is type function returning pointer to integer
`int (*)()`	is type pointer to function returning an integer
`int (*[3])()`	is type array of three pointers to functions returning an integer

Typedef

Declarations whose storage class is `typedef` do not define storage, but instead define identifiers. Your program can later use these identifiers as if they were type keywords naming fundamental or derived types.

typedef-name:

identifier

Within a declaration that involves `typedef`, each identifier that is part of a declarator is syntactically equivalent to the type keyword that names the identifier type as described in "Meaning of Declarators," earlier in this chapter. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct {double re, im;} complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the following types apply:

- `distance` is `int`
- `metricp` is a *pointer to* `int`
- `z` is the specified structure `complex`
- `zp` is a *pointer to such a structure*

The `typedef` does not introduce brand new types, only synonyms for types can be specified in another way. Thus, in the example above, `distance` is considered to have exactly the same type as any other `int` object.

Statements

Except as indicated, statements are executed in sequence.

Expression statement

Most statements are expression statements, which have the form

expression;

Usually expression statements are assignments or function calls.

Compound statement or block

The compound statement lets your program use several statements where only one is expected:

compound-statement:

{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:

declaration

declaration declaration-list

statement-list:

statement

statement statement-list

If any of the identifiers in the *declaration-list* were declared previously, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. Although it is poor practice, your program can transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once, when the program begins execution. Inside a block, `extern` declarations do not reserve storage, so initialization is not permitted.

Conditional statement

The two forms of the conditional statement are

`if (`*expression*`)` *statement*

`if (`*expression*`)` *statement* `else` *statement*

In both cases the expression is evaluated. If it is nonzero, the first substatement is executed. If the expression is zero, the second substatement is executed. The "else" ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

`while` statement

The `while` statement has the form

`while (`*expression*`)` *statement*

The substatement is executed repeatedly as long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

`do` statement

The `do` statement has the form

`do` *statement* `while (`*expression*`);`

The substatement is executed repeatedly until the value of the expression is zero. The test takes place after each execution of the statement.

`for` statement

The `for` statement has the form

`for (`*exp-1*$_{opt}$`;` *exp-2*$_{opt}$`;` *exp-3*$_{opt}$`) statement`

This statement is equivalent to

exp-1$_{opt}$ `;`
`while (`*exp-2*$_{opt}$`)`
`{`
statement
exp-3$_{opt}$`;`
`}`

except in the case where a `continue` appears before or in *exp-3*. In this case, (all of) *exp-3* is not read or implemented (see "`continue` Statement").

The first expression specifies initialization for the loop; the second specifies a test made before each iteration, so that the loop is exited when the expression becomes zero. The third expression often specifies an incrementation that is performed after each iteration.

Any or all of the expressions can be dropped. A missing *exp-2* makes the implied `while` clause equivalent to `while(1)`. Other missing expressions are simply dropped from the expansion above.

`switch` statement

The `switch` statement causes control to be transferred to one of several statements, depending on the value of an expression. It has the form

`switch (`*expression*`)` *statement*

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement *within* the statement can be labeled with one or more case prefixes, as in

`case` *constant-expression*`:`

where the constant expression must be `int`. No two case constants in the same `switch` can have the same value. Constant expressions are precisely defined in "Constant Expressions," later in this chapter.

There also can be no more than one statement prefix of the form

```
default:
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the expression value, control is passed to the statement following the matched case prefix. If no case constant matches the expression, control passes to the statement with the `default` prefix. If no case matches and there is no `default`, none of the statements in the switch are executed.

The prefixes `case` and `default` do not alter the flow of control; it continues unimpeded across such prefixes. To learn about exiting from a switch, see the next section, "`break` Statement."

Usually, the statement that is the subject of a switch is compound. Declarations can appear at the head of this statement, but initializations of automatic or register variables are ineffective.

`break` statement

The statement

```
break;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement. Control passes to the statement following the terminated statement.

`continue` statement

The statement

```
continue;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement—that is, to the end of the loop.

More precisely, in each of the following statements,

Statement 1:

```
while (exp-1) {
        exp-2
contin:;
}
```

Statement 2:

```
do {
        exp-1
contin:;
} while (exp-2);
```

Statement 3:

```
for (exp-1) {
        exp-2
contin:;
}
```

a `continue` is equivalent to `goto contin` (following the `contin:` is a null statement; see "Null Statement," later in this chapter).

`return` statement

A function returns to its caller by means of the `return` statement, which has one of the two forms

```
return;
return expression;
```

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a `return` with no returned value. The *expression* can be enclosed in parentheses.

`goto` statement

Control can be transferred unconditionally by means of the statement

`goto` *identifier*;

The identifier must be a label (see the next section, "Labeled Statement") located in the current function.

Labeled statement

Any statement can be preceded by label prefixes of the form

```
identifier:
```

which serve to declare the identifier as a label. The only use of a label is as a target of a `goto`. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared (see "Scope Rules," later in this chapter).

Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the ending brace of a compound statement or to supply a null body to a looping statement such as `while`.

External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type specifier (see "Type Specifiers," earlier in this chapter) also can be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared, just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as for all declarations, except that only at this level can the code for functions be given.

External function definitions

Function definitions have the form

function-definition:

decl-specifiers$_{opt}$ *function-declarator function-body*

The only storage class specifiers allowed among the declaration specifiers are `extern` or `static` (see "Scope of Externals," later in this chapter, for the distinction between them). A function declarator is similar to a declarator for a

function returning some-type

except that it lists the formal parameters of the function being defined.

function-declarator:

declarator (*parameter-list*$_{opt}$)

parameter-list:

identifier

identifier, *parameter-list*

The function-body has the form

function-body:

declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, can be declared in the declaration list. Any identifier whose type is not given is taken to be `int`. The only storage class that can be specified is `register`; if it is specified, the corresponding actual parameter is copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
            int a, b, c;
{
            int m;

            m = (a > b) ? a : b;
            return((m > c) ? m : c);
}
```

Here, `int` is the *type-specifier;* `max(a, b, c)` is the *function-declarator;* `int a, b, c;` is the *declaration-list* for the formal parameters, and `{...}` is the *block* giving the code for the statement.

The C compiler converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`.

All `char` and `short` formal parameter declarations are similarly adjusted to read `int`. Also, because a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared

array of some-type

are adjusted to read

pointer to some-type

External data definitions

An external data definition has the form

data-definition:

declaration

The storage class of such data can be `extern` (the default) or `static`, but not `auto` or `register`.

Scope rules

A C program doesn't have to be compiled entirely at the same time. The source text of the program can be kept in several files and precompiled routines can be loaded from libraries. Communication among the functions of a program can be carried out through both explicit calls and manipulation of external data.

Therefore, there are two kinds of scope to consider: (1) **lexical scope,** which is essentially the region of a program within which your program can use some identifier without drawing "undefined identifier" diagnostics, and (2) **scope of externals,** which is the scope associated with external identifiers; it is characterized by the rule that states that references to the same external identifier are references to the same object.

Lexical scope

The lexical scope of identifiers that are declared in external definitions persists from the definition through the end of the source file in which they appear.

The lexical scope of identifiers that are formal parameters persists through the function with which they are associated.

The lexical scope of identifiers that are declared at the head of a block persists until the end of the block.

The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes that do not conflict (see "Structure and Union Declarations" and "Enumeration Declarations," earlier in this chapter). Members and tags follow the same scope rules as other identifiers.

The `enum` constants are in the same class as ordinary variables and follow the same scope rules.

The `typedef` names are in the same class as ordinary identifiers. They can be redeclared in inner blocks, but an explicit type must be given in the inner declaration.

```
typedef float distance;
...
{
            auto int distance;
            ...
```

The `int` must be present in the second declaration or it is taken as a declaration with no declarators and with type `distance`.

Scope of externals

If a function refers to an identifier that's declared to be `extern`, somewhere among the files or libraries that constitute the complete program there must be at least one external definition for that identifier. All functions in a given program that refer to the same external identifier are referring to the same object, so you must take care that the type and size you specify in the definition are compatible with those specified by each function that references the data.

It is illegal to initialize any external identifier explicitly more than once in the set of files and libraries that make up a multifile program. Your program can have more than one data definition for any external nonfunction identifier, however; explicit use of `extern` does not change the meaning of an external declaration.

With a more restrictive compiler, the use of the `extern` storage class takes on an additional meaning. With such a compiler, the explicit appearance of the `extern` keyword in the external data declarations of identities without initialization indicates that the storage of the identifier is allocated elsewhere, either in that file or in another file. Your program must have exactly one definition of each external identifier (without `extern`) in the set of files and libraries composing a multifile program.

The A/UX C compiler accepts multiply defined externals. For future portability of code, however, you might find it easier to observe the above restrictions in any case. To help you do this, you can use the `-M` flag option to `ld`, which causes the link editor to check for multiply defined externals. (The flag option should be entered on the `cc` command line and is passed to `ld` by `cc`.) `ld` prints a warning message if any multiple definitions are found.

In addition, in A/UX, `ld` warns you, by default, if the size of these **multiple externs** differs among the files in which it is found. This catches such errors as a variable defined as `char` in one file and as `int` in another. You can use the `-t` flag option to `ld` to disable this check. To invoke this option on the `cc` command line, you must pass it explicitly to `ld` through the `-W` option to `cc`, as

```
cc -Wl-t
```

where `-W` passes an argument to the link editor (`l`), and `-t` is the argument passed to `ld`. This form must be used, as the `-t` option to `cc` is already defined to mean something else.

Together, the `-M` and `-t` flag options to `ld` allow for simulation of the more restrictive environment required by other machines. Using these options, you might find it easier to write code that ports to more restrictive compilers with fewer, if any, changes.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions can be declared `static`. This provides a way of hiding globals, and hence should be used with caution.

Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with `#` communicate with this preprocessor. There can be any number of blanks and horizontal tabs between the `#` and the directive. These lines have syntax independent of the rest of the language; they can appear anywhere. Their effect lasts (independent of scope) until the end of the source program file.

Token replacement

A compiler-control line of the form

`#define` *identifier token-string*

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token string are taken as part of that string. A line of the form

`#define` *identifier* `(`*identifier*, `...)` *token-string*

where there is no space between the first identifier and the (is a macro definition with arguments. It can have zero or more formal parameters. Subsequent instances of the first identifier, followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call.

The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or commas protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the *token-string* are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers for replacement.

In both forms, the replacement string is rescanned for more defined identifiers. In both forms, a long definition can be continued on another line by preceding the newline with a backslash (`\`). This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

`#undef` *identifier*

causes the identifier preprocessor (if any) to be dropped.

If a `#define` identifier is the subject of a subsequent `#define` with no intervening `#undef`, the two token strings are compared textually. If the two token strings are not identical (all white space is considered equivalent), the identifier is considered to be redefined.

File inclusion

A compiler control line of the form

`#include "`*filename*`"`

causes that line to be replaced by the entire contents of the file *filename*. The named file is first searched for in the directory of the file containing the `#include`, and then in a sequence of specified or standard places. Alternatively, a control line of the form

`#include <`*filename*`>`

searches only the specified or standard places and not the directory of the `#include` (how the places are specified is not part of the language). `#include` statements can be nested.

Conditional compilation

A compiler control line of the form

`#if` *restricted-constant expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "Constant Expressions," later in this chapter. Here, the restricted-constant expression cannot contain `sizeof` casts or an enumeration constant.)

A restricted-constant expression also can contain the additional unary expression

`defined` *identifier*

or

`defined(`*identifier*`)`

each of which evaluates to one if the identifier is currently defined in the preprocessor, and to zero if it is not.

All currently defined identifiers in restricted-constant expressions are replaced by their token strings (except those identifiers modified by `defined`), just as in normal text. The restricted-constant expression is evaluated only after all expressions are finished. During this evaluation, all identifiers undefined to the procedure evaluate to zero.

A control line of the form

`#ifndef` *identifier*

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a `#define` control line. It is equivalent to `#if!def` `(`*identifier*`)`.

A control line of the form

`#ifndef` *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to `#if!defined (`*identifier*`)`.

All three forms are followed by an arbitrary number of lines that can include the control line

`#else`

followed by the control line

`#endif`

If the checked condition is true, any lines between `#else` and `#endif` are ignored. If the checked condition is false, any lines between the test and `#else` or, lacking `#else`, `#endif`, are ignored.

These constructions can be nested.

Line control

For the benefit of other preprocessors that generate C programs, a line of the form

`#line` *constant filename*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by *filename*. If *filename* is absent, the remembered filename does not change.

Implicit declarations

When you are writing a program, you don't always have to specify both the storage class and type of identifiers in a declaration. The storage class is supplied by the context in external definitions, declarations of formal parameters, and structure members. In a declaration inside a function, if you specify a storage class but no type, the identifier is assumed to be `int`. If you specify a type but no storage class, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, because `auto` functions do not exist. If the type of an identifier is

function returning some-type

it is implicitly declared to be `extern`.

In an expression, an undeclared identifier followed by `(` is contextually declared to be *function returning* `int`.

Types revisited

This section summarizes the operations that can be performed on objects of certain types.

Structures and unions

Structures and unions can be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the `->` or dot (`.`) must specify a member of the aggregate that is named or pointed to by the expression on the left. In general, a member of a union cannot be inspected unless that member has a value assigned more recently than any other member that overlaps the same space. One special guarantee is made by the language, however, to simplify the use of unions: If a union contains several structures that share a common initial sequence and the union currently contains one of these structures, you can inspect the common part of any member in which it occurs. For example, the following is a legal fragment.

```
union
{
        struct
        {
                int             type;
        } n;
        struct
        {
                int             type;
                int             intnode;
        } ni;
        struct
        {
                int             type;
                float           floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

Functions

A program can do only two things with a function: call it or take its address. If the name of a function appears in an expression, not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, your program can include

```
int f();
...
g(f);
```

The definition of `g` can read

```
g(funcp)
int (*funcp)();
{
          ...
          (*funcp)();
          ...
}
```

Notice that `f` must be declared explicitly in the calling routine because its appearance in `g(f)` was not followed by `(`.

Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator `[ ]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules that apply to `+`, if `E1` is an array and `E2` an integer, `E1[E2]` refers to the `E2`th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If `E` is an *n*-dimensional array of rank *i*x`j`x...x*k*, then `E` appearing in an expression is converted to a pointer to an (*n*-1)-dimensional array with rank `jx...xk`. If the `*` operator is applied to this pointer, either explicitly or implicitly as a result of subscripting, the result is the pointed-to (*n*-1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here `x` is a 3x5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression `x[i]`, which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `i` is converted to the type of `x`, which involves multiplying `i` by the length of the object to which the pointer points, namely, five-integer objects.

The results are added and indirection applied to yield an array (of five integers), which, in turn, is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored by rows (last subscript varies most quickly). The first subscript in the declaration helps determine the amount of storage consumed by an array, but plays no other part in subscript calculations.

Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator; see "Unary Operators" and "Type Names," both earlier in this chapter.

A pointer can be converted to any of the integral types large enough to hold it. Whether an `int` or `long` is required is machine-dependent. The mapping function is also machine-dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for this machine are given below.

An object of integral type can be converted explicitly to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine-dependent.

A pointer to one type can be converted to a pointer to another type. The resulting pointer can cause addressing exceptions upon use, if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size can be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a `char` pointer,

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The `alloc` must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to `double`; then the use of the function is portable.

In A/UX, pointers are 32 bits long and measure bytes. This is the same size as an `int` or `long`. The `char` values have no alignment requirements; everything else must have an even address.

Constant expressions

In several places C requires expressions that evaluate to a constant:

- after `case`
- as array bounds
- in initializers

In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and `sizeof` expressions, possibly connected by the binary operators,

```
=  -  *  /  %  &  |  ^
<< >> == != < > <= >= && ||
```

or by the unary operators,

```
-  ~
```

or by the ternary operators,

```
?:
```

Parentheses can be used for grouping, but not for function calls.

When writing your program, you have more latitude with initializers. In addition to constant expressions, you also can use floating constants and arbitrary casts. You also can apply the unary `&` operator to external or static objects and to external or static arrays subscripted with a constant expression. You can apply the unary `&` implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object—plus or minus a constant.

Portability considerations

Certain parts of C are inherently machine-dependent. The following description of potential trouble spots is meant not to be complete, but to point out the main ones.

Purely hardware issues like word size and the properties of floating-point arithmetic and integer division have not proven to be problems. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most others are only minor problems.

The number of `register` variables that actually can be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machines; excess or invalid `register` declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Because character constants are really objects of type `int`, multicharacter constants can be permitted. The specific implementation is machine-dependent, because the order in which characters are assigned to a word varies from one machine to another. (See "Character Constants," earlier in this chapter, for the treatment of multicharacter character constants on the 68020.)

Fields are assigned to words, and characters to integers, from right to left on some machines and from left to right on other machines. (Bit fields run from left to right in this implementation.) These differences are invisible to isolated programs that do not indulge in **type punning** (that is, by converting an `int` pointer to a `char` pointer and inspecting the storage pointed to), but must be accounted for when conforming to externally imposed storage layouts.

Syntax summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

Expressions

The basic expressions are

```
expression:
        primary
        * expression
        & lvalue
        - expression
        ! expression
        ~ expression
        ++ lvalue
        -- lvalue
        lvalue ++
        lvalue --
        sizeof expression
        sizeof (type-name)
        (type-name) expression
        expression binop expression
        expression ? expression : expression
        lvalue asgnop expression
        expression, expression
```

primary:

```
            identifier
            constant
            string
            (expression)
            primary (expression-listopt)
            primary [expression]
            lvalue . identifier
            primary -> identifier
```

lvalue:

```
            identifier
            primary [expression]
            lvalue . identifier
            primary -> identifier
            * expression
            (lvalue)
```

The primary-expression operators

```
()   []   .   ->
```

have highest priority and group left to right. The unary operators

```
*  &  -  !  ~  ++  --  sizeof  ( type-name )
```

have priority below the primary operators but above any binary operator and group right to left. Binary operators group left to right; they have decreasing priority, as shown here:

binop:

```
    *      /      %
    +      -
    >>     <<
    <      >      <=     >=
    ==     !=
    &
    ^
    |
    &&
    ||
```

The conditional operator groups right to left. Assignment operators all have the same priority and all group right to left.

asgnop:

```
        =   +=   -=   *=   /=   %=
        >>=   <<=   &=   ^=   |=
```

The comma operator has the lowest priority and groups left to right.

Declarations

declaration:

    decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:

    type-specifier decl-specifiers$_{opt}$
    sc-specifier decl-specifiers$_{opt}$

sc-specifier:

```
auto
static
extern
register
typedef
```

type-specifier:

    basic-type-specifier
    struct-or-union-specifier
    typedef-name
    enum-specifier

basic-type-specifier:

    basic-type
    basic-type basic-type-specifiers

basic-type

`char`

`short`

`int`

`long`

`unsigned`

`float`

`double`

enum-specifier:

`enum {`*enum-list*`}`

`enum` *identifier* `{`*enum-list*`}`

`enum` *identifier*

enum-list:

enumerator

enum-list`,` *enumerator*

enumerator:

identifier

identifier `=` *constant-expression*

init-declarator-list:

init-declarator

init-declarator`,` *init-declarator-list*

init-declarator:

declarator *initializer*$_{opt}$

declarator:

identifier

`(`*declarator*`)`

`*` *declarator*

declarator `()`

declarator `[`*constant-expression*$_{opt}$`]`

struct-or-union-specifier:

`struct` { *struct-decl-list* }
`struct` *identifier* { *struct-decl-list* }
`struct` *identifier*
`union` { *struct-decl-list* }
`union` *identifier* { *struct-decl-list* }
`union` *identifier*

struct-decl-list:

struct-declaration
struct-declaration *struct-decl-list*
struct-declaration:
type-specifier *struct-declarator-list*;

struct-declarator-list:

struct-declarator
struct-declarator, *struct-declarator-list*
struct-declarator:
declarator
declarator: *constant-expression*
: *constant-expression*

initializer:

= *expression*
= { *initializer-list* }
= { *initializer-list*, }

initializer-list:

expression
initializer-list, *initializer-list*
{ *initializer-list* }
{ *initializer-list*, }

type-name:

type-specifier *abstract-declarator*

abstract-declarator :

  empty
  (*abstract-declarator*)
  `*` *abstract-declarator*
  abstract-declarator `()`
  abstract-declarator [*constant-expression*$_{opt}$]

typedef-name:

  identifier

Statements

compound-statement:

  {*declaration-list*$_{opt}$ *statement-list*$_{opt}$}

declaration-list:

  declaration
  declaration declaration-list

statement-list:

  statement
  statement statement-list
  statement:
  compound-statement
  expression ;
  `if` (*expression*) *statement*
  `if` (*expression*) *statement* `else` *statement*
  `while` (*expression*) *statement*
  `do` *statement* `while` (*expression*) ;
  `for` (*exp*$_{opt}$; *exp*$_{opt}$; *exp*$_{opt}$) *statement*
  `switch` (*expression*) *statement*
  `case` *constant-expression*: *statement*
  `default:` *statement*
  `break;`

`continue;`

`return;`

`return` *expression*`;`

`goto` *identifier*`;`

`identifier:` *statement*

`;`

External definitions

program:

external-definition

external-definition program

external-definition:

function-definition

data-definition

function-definition:

type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:

declarator `(`*parameter-list*$_{opt}$`)`

parameter-list:

identifier

identifier`,` *parameter-list*

function-body:

`{`*declaration-list*$_{opt}$ *compound-statement*`}`

data-definition:

`extern`$_{opt}$ *declaration*`;`

`static`$_{opt}$ *declaration*`;`

Preprocessor

`#define` *identifier token-string*

`#define` *identifier* `(`*identifier*`, ...)` *token-string*

`#undef` *identifier*

`#include "`*filename*`"`

`#include <`*filename*`>`

`#if` *restricted-constant-expression*

`#ifdef` *identifier*

`#ifndef` *identifier*

`#else`

`#endif`

`#line` *constant* `"`*filename*`"`

4 C Implementation Notes

This chapter describes the A/UX C programming language, including how data is represented, how data is passed between functions, the environment of a function, and the calling mechanism for a function. The information in this chapter is intended for programmers who must have detailed knowledge of the interface mechanisms to match C code with the assembler. It is also intended for those who wish to write new system functions or mathematical functions.

When a C program is compiled and assembled, the program is split into three parts:

`text`	The executable code of the program; the compiler/assembler combination produces this.
`data`	The initialized data area; this contains literal constants, character strings, and so on. The compiler/assembler combination produces this.
`bss`	The uninitialized data areas; the loader generates and clears this area to zero at load time. This is a feature of the system and can be relied upon.

During execution of a program, the stack area contains indeterminate data. In other words, its previous contents (if any) cannot be relied upon.

Data representations

In general, all data elements of whatever size are stored so that their least significant bit is in the highest addressed byte and their most significant bit is in the lowest addressed byte. The list below describes the representation of data:

`char`	Values of type `char` occupy 8 bits. Such values can be aligned on any byte boundary.
`short`	Values of type `short` occupy 16 bits. Values of type `short` are aligned on word (16-bit) address boundaries.
`long`	Values of type `long` occupy 32 bits. A `long` value is the same as an `int` value in 68020 C. Values of this type are aligned on word (16-bit) address boundaries.
`float`	Values of type `float` occupy 32 bits. All `float` values are automatically converted to type `double` for computation purposes, except when testing for zero or nonzero. Values of this type are aligned on word (16-bit) boundaries. A `float` value consists of a sign bit, followed by an 8-bit biased exponent, followed by a 23- bit mantissa (24 bits including the hidden bit). Values of type `float` are stored in IEEE Floating Point Standard P754 representation.
`double`	Values of type `double` occupy 64 bits. Values of this type are aligned on word (16-bit) boundaries. A `double` value consists of a sign bit, followed by an 11-bit biased exponent, followed by a 52-bit mantissa (53 bits including the hidden bit). Values of type `double` are stored in IEEE representation.
pointer	All *pointers* are represented as long (32-bit) values. Pointers are aligned on word (16-bit) boundaries.
array	The base address of an *array* value is always aligned on a word (16-bit) address boundary. Elements of an *array* are stored contiguously, one after the other. Elements of multidimensional arrays are stored in row-major order. That is, the last dimension of an array varies the most quickly. When a multidimensional array is declared, it is possible to omit the size specification for the last dimension. In such a case, what is allocated is actually an array of pointers to the elements of the last dimension.

`struct` and `union` Within structures and unions, it is possible to obtain unfilled holes of size `char`. This is because the compiler rounds addresses up to 16-bit boundaries to accommodate word-aligned elements.

This situation can best be demonstrated by an example. Consider the following structure:

```
struct   {
        int   x;          /* This is a 32-bit element */
        char  y;          /* Takes up a single byte */
        short z;          /* Aligned on 16-bit boundary */
};
```

The total number of bytes declared above is seven: four for the `int`, one for the `char`, and two for the `short`.

In reality, the `z` field, which is a `short`, is aligned on a 16-bit boundary by the C compiler. In this case, the compiler inserts a hole after the `char` element `y` to align the `short` element `z`. The net effect of these machinations is a structure that behaves like this:

```
struct {
        int   x;          /* This is a 32-bit element */
        char  y;          /* Takes up a single byte */
        char  dummy;      /* Fills the structure */
        short z;          /* Aligned to a 16-bit boundary */
};
```

The C compiler never reorders any parts of a structure. Similar considerations apply to arrays of structures or unions. Each element of an array (other than an array of `char`) begins on a 16-bit boundary.

For a detailed treatment of data storage, consult *The C Programming Language* by Kernighan and Ritchie.

Parameter passing in C

The C programming language is unique in that it really has only functions. The effect of a subroutine is achieved simply by having a function that does not return a value. The type of such a function should be `void`.

Another unique feature of C is that parameters to functions are always passed by value. The C programming language has no concept of declaring parameters to be passed by reference, as in languages such as Pascal. To pass a parameter by reference in a C program, the programmer must pass the address of the parameter explicitly. The called function must be aware that it is receiving an address instead of a value, and the appropriate code must be present to handle that case.

When a function is called, its parameters (if any) are evaluated and then pushed onto the stack in reverse order. All parameters are pushed onto the stack as 32-bit `longs`, except for `floats` and `doubles`, which are pushed as 64-bit `doubles`. If a parameter is shorter than 32 bits, it is expanded to a 32-bit value with sign extension, if necessary. The calling procedure is responsible for popping the parameters off the stack.

Consider a C function call such as

```
ferry (charon, 7, &styx, 1<<10);
```

After parameter evaluation, but just before the call, the stack looks like this:

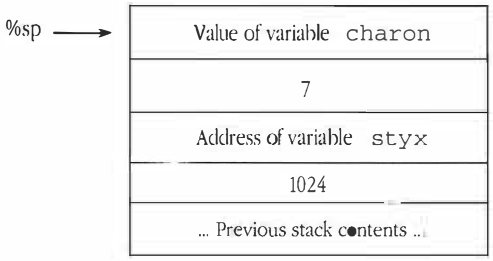

Figure 4-1 Stack contents after evaluation of function call

Functions are called by issuing either a `bsr` instruction or a `jsr` instruction, depending on whether the callee is within a 16-bit addressing range or not and whether the C optimizer was used. The `bsr` or `jsr` instruction pushes the return address onto the stack and then branches to the indicated function. After the call, on entry to the function, the stack looks like Figure 4-2.

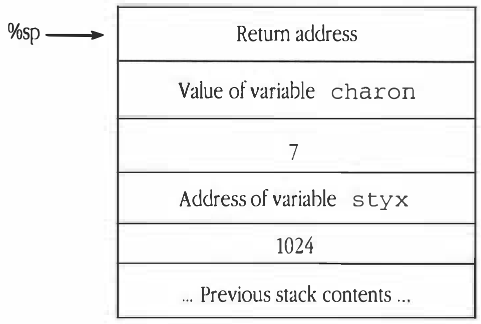

Figure 4-2 Stack contents after entry to the function call

In each function, register `%a6` is used as a stack frame base. The stack location referenced by `%a6` contains the return address.

Setting up the stack

Upon entry into the function, the prolog code is executed. The **prolog code** allocates sufficient space on the stack for the local variables, plus sufficient space to save any registers that this function uses. The prolog code looks like this:

```
link.l      %fp,&F%1

movm.l      &M%1,(4,%sp)
```

The `F%1` constant is the size of the stack frame for the local variables, plus 4 bytes for each ordinary register variable and 12 bytes for each `float` or `double` register variable.

The `M%1` constant is a mask to determine which registers need to be saved on the stack for this particular function. This is dependent on the register variables that the programmer declared for that particular routine. If the function has floating-point register variables, the `movm.l` instruction is followed by

```
fmovm &FPM%1, (FPO%1, %sp)
```

which saves the floating-point registers used by the routine for register variables of types `float` and `double`. `FPO%1` is the offset of the floating register save area, and `FPM%1` is a mask to tell the `fmovm` instruction which registers to save.

Allocating local variables and registers

A total of ten registers are available for register variables. Six of these are data (`%d`) registers, and four are address (`%a`) registers. The available `%a` registers are `%a2` through `%a5`. The available `%d` registers are `%d2` through `%d7`. There are also six floating-point registers on the 68881 (`%fp2` through `%fp7`) available for register variables of type `float` and `double`.

The location of a function return value depends on the type of the function. Functions that return integral types (`char`, `short`, `int`, `long`, or the `unsigned` versions of any of these) return their results in `%d0`. Functions returning pointers return their results in `%a0`, while `float` and `double` functions use `%fp0`. Structure-valued and union-valued functions return their results in `%d0` if the entire `struct` or `union` fits in 32 bits; otherwise, the return value is stored in a special temporary area inside the function, a pointer to this temporary area is returned in `%a0`, and if the return value is used, code is generated to copy the returned `struct` or `union` into the appropriate place.

Remember that undeclared functions are assumed to be of type `int`. It follows that functions must be declared if they return values of type `float`, `double`, `pointer`, `struct`, or `union`, or else the generated code is wrong. Use the `lint` program to find places where functions are not declared (see Chapter 8, "`lint` Reference").

`pointer` register variables are assigned only to address registers, `float` and `double` register variables only to floating-point registers. Other register variables are assigned only to data registers. Register declarations are ignored for variables of type `struct` or `union`.

Register variables are allocated to registers in the order in which they are declared in the C source program, starting at the low end (`%a2`, `%d2`, or `%fp2`) of the appropriate type of register.

If there are more register variables of either kind than there are registers to accommodate them, the remaining variables are allocated on the stack as local variables, just as if the register attribute had never been given in the declaration.

When the prolog code is completed, the stack looks like Figure 4-3.

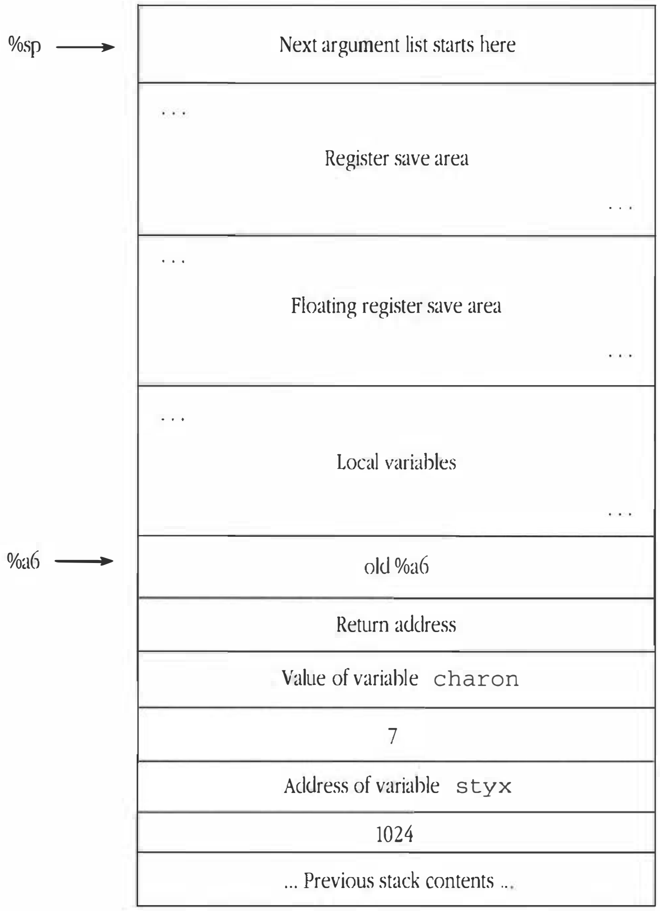

Figure 4-3 Stack contents after executing prolog code

Returning from a function or subroutine

Upon reaching a `return` statement, either explicit or implicit, the function executes the **epilog code.** If the function has a return value, it is generated from the line

`return(`*expression*`);`

The value of *expression* (converted, if necessary, to match the type of the function) is placed in register `%d0`, `%a0`, or `%fp0`, as appropriate, and the epilog code is executed to effect a return from the function. The epilog code looks like this:

```
movm.l        (4,%sp),     &M%1
unlk                       %fp
rts
```

The `movm.l` instruction restores any registers that were saved during the prolog. If there were floating-point register variables, the `movm.l` instruction is followed by

```
fmovm (FPO%1, %sp), &FPM%1
```

which restores the floating-point registers that were saved. The stack frame base pointer in `%fp` is then put back to the point where `%fp` once again points to the return address, and the function is exited through the `rts` instruction, which pops the stack to the state it was in prior to the original call and returns to the function that called it.

System calls

The C compiler generates code for system calls by calling library routines that place the system call number in register `%d0` and execute a `TRAP &0` instruction.

Parameters are passed on the user stack in the C calling convention. On return from the system call, errors are signaled by the carry flag being set. The C interface to the system calls typically returns a `-1` on error, as the carry flag cannot be tested from C.

Optimizations

The C compiler can be run to optimize the code it generates, making that code both compact and fast. The command line

`cc -O` *file*

generates optimized code.

Use of register variables

The decision to declare a variable in a register should depend on the number of times that variable is referenced during the execution of a function. If a variable is used more than twice in a function, it can be declared as a register variable. If a variable is used less than twice in a function, it is not useful to declare it as a register variable, because the amount of time spent saving and restoring that register is more than the time saved in using a register instead of a location on the stack.

Miscellaneous notes

The object files created by the assembler and linker use the common object file format (see Chapter 16, "COFF Reference").

The C compiler accepts multiply defined external variables, as long as no more than one of the definitions includes an initialization.

The C compiler supports floating and double variables by using the 68881. Floating-point data values are represented in IEEE standard floating-point format.

5 The Standard C Library (`libc`)

This chapter describes the A/UX C library. A **library** is a collection of related functions and declarations. Using a library simplifies programming efforts by linking what is needed, allowing use of locally produced functions, and so on. All the functions described in this chapter are also described in Section 3 of *A/UX Programmer's Reference*. Most of the declarations described in this chapter are also described in Section 5 of *A/UX Programmer's Reference*.

This C library is the basic library for C language programs. The C library is made up of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described in greater detail later in this chapter.

Including functions

The C library is made up of several types of functions. When a program is being compiled, the compiler automatically searches the C language library to locate and include functions that are used in the program. All C library functions are loaded automatically by the compiler, although you must sometimes include the proper header files, with its various declarations in your program, for the functions to work properly. C library functions are divided into the following types:

- input/output control
- string manipulation
- character manipulation
- time functions
- miscellaneous functions

Including declarations

Some functions need a set of declarations to operate properly. A set of declarations is stored in a file called a **header file** (with a `.h` extension). Header files for the C library are stored in the `/usr/include` directory. To include a certain header file in your program, you must specify the following near the top of the file containing the program:

`#include <`*file*`.h>`

where *file*`.h` is the name of the header file. Because the header files define the type of functions and various preprocessor constants, you must include them before invoking the functions they declare.

Input/output control

C library functions are automatically included as needed during the compiling of a C language program. No command line request is needed.

You must include the header file required by the input/output functions near the beginning of each file that references an input or output function:

```
#include <stdio.h>
```

The input/output functions are grouped into the following categories:

- file access
- file status
- input
- output
- miscellaneous

File access functions

Function	*Reference*	*Brief description*
`fclose`	`fclose`(3S)	close an open stream
`fdopen`	`fopen`(3S)	associate stream with a file opened with the `open`(2) system call
`fileno`	`ferror`(3S)	file descriptor associated with an open stream
`fopen`	`fopen`(3S)	open a file with specified permissions and return a pointer to a stream that is used in subsequent references to the file
`freopen`	`fopen`(3S)	substitute named file in place of open stream
`fseek`	`fseek`(3S)	reposition the file pointer
`pclose`	`popen`(3S)	close a stream opened by `popen`
`popen`	`popen`(3S)	create pipe as a stream between calling process and command
`rewind`	`fseek`(3S)	reposition file pointer at beginning of file

`setbuf`	`setbuf`(3S)	assign buffering to stream
`setvbuf`	`setbuf`(3S)	similar to `setbuf`, but allowing finer control

File status functions

Function	*Reference*	*Brief description*
`clearerr`	`ferror`(3S)	watch for side effects; reset error condition on stream
`feof`	`ferror`(3S)	watch for side effects; test for end-of-file (EOF) on stream
`ferror`	`ferror`(3S)	watch for side effects; test for error condition on stream
`ftell`	`fseek`(3S)	return current position in the file

Input functions

Function	*Reference*	*Brief description*
`fgetc`	`getc`(3S)	true function for `getc`(3S)
`fgets`	`gets`(3S)	read string from stream
`fread`	`fread`(3S)	general buffered read from stream
`fscanf`	`scanf`(3S)	formatted read from stream
`getc`	`getc`(3S)	watch for side effects; read character from stream
`getchar`	`getc`(3S)	watch for side effects; read character from standard input
`gets`	`gets`(3S)	read string from standard input
`getw`	`getc`(3S)	read word from stream
`scanf`	`scanf`(3S)	read using format from standard input
`sscanf`	`scanf`(3S)	formatted read from a string
`ungetc`	`ungetc`(3S)	put back one character on stream

Output functions

Function	*Reference*	*Brief description*
`fflush`	`fclose`(3S)	write all currently buffered characters from stream
`fprintf`	`printf`(3S)	formatted write to stream
`fputc`	`putc`(3S)	true function for `putc` (3S)
`fputs`	`puts`(3S)	write string to stream
`fwrite`	`fread`(3S)	general buffered write to stream
`printf`	`printf`(3S)	print using format to standard output
`putc`	`putc`(3S)	watch for side effects; write character to standard output
`putchar`	`putc`(3S)	watch for side effects; write character to standard output
`puts`	`puts`(3S)	write string to standard output
`putw`	`putc`(3S)	write word to stream
`sprintf`	`printf`(3S)	formatted write to string
`vfprintf`	`vprint`(3C)	print using format to stream by `varargs`(3X) argument list
`vprintf`	`vprint`(3C)	print using format to standard output by `varargs`(3X) argument list
`vsprintf`	`vprintf`(3C)	print using format to stream string by `varargs`(3X) argument list

Miscellaneous functions

Function	*Reference*	*Brief description*
`ctermid`	`ctermid`(3S)	return filename for controlling terminal
`cuserid`	`cuserid`(3S)	return login name for owner of current process
`system`	`system`(3S)	execute shell command
`tempnam`	`tempnam`(3S)	create temporary filename using directory and prefix
`tmpnam`	`tmpnam`(3S)	create temporary filename
`tmpfile`	`tmpfile`(3S)	create temporary file

String manipulation functions

These functions are used to locate characters within a string or to copy, concatenate, or compare strings. These functions are automatically located and loaded during the compiling of a C language program. No command-line request is needed because these functions are part of the C library. The string manipulation functions are declared in a header file that you should include near the beginning of each file that uses any of these functions:

```
#include <string.h>
```

Function	*Reference*	*Brief description*
`strcat`	`string`(3C)	concatenate two strings
`strchr`	`string`(3C)	search string for character
`strcmp`	`string`(3C)	compares two strings
`strcpy`	`string`(3C)	c●py string
`strcspn`	`string`(3C)	length of initial string not containing set of characters
`strlen`	`string`(3C)	length of string
`strncat`	`string`(3C)	concatenate two strings with a maximum length
`strncmp`	`string`(3C)	compare two strings with a maximum length
`strncpy`	`string`(3C)	copy string over string with a maximum length
`strpbrk`	`string`(3C)	search string for any set of characters
`strrchr`	`string`(3C)	search string backward for character
`strspn`	`string`(3C)	length of initial string containing set of characters
`strtok`	`string`(3C)	search string for token separated by any of a set of characters

Character manipulation

The following functions and declarations are used for testing and translating ASCII characters. These functions are located and loaded automatically during the compiling of a C language program. No command-line request is needed because these functions are part of the C library.

You should include the declarations associated with these functions near the beginning of the file being compiled:

```
#include <ctype.h>
```

Character testing functions

These functions can be used to identify characters as uppercase or lowercase letters, digits, punctuation, and so on:

Function	*Reference*	*Brief description*
`isalnum`	`ctype`(3C)	return true if character is alphanumeric
`isalpha`	`ctype`(3C)	return true if character is alphabetic
`isascii`	`ctype`(3C)	return true if integer is an ASCII character
`iscntrl`	`ctype`(3C)	return true if character is a control character
`isdigit`	`ctype`(3C)	return true if character is a digit
`isgraph`	`ctype`(3C)	return true if character is a printable character
`islower`	`ctype`(3C)	return true if character is a lowercase letter
`isprint`	`ctype`(3C)	return true if character is a printing character including space
`ispunct`	`ctype`(3C)	return true if character is a punctuation character
`isspace`	`ctype`(3C)	return true if character is a white space character
`isupper`	`ctype`(3C)	return true if character is an uppercase letter
`isxdigit`	`ctype`(3C)	return true if character is a hex digit

Character translation functions

These functions provide translation of uppercase to lowercase, lowercase to uppercase, and integer to ASCII:

Function	*Reference*	*Brief description*
`toascii`	`conv`(3C)	convert integer to ASCII character
`tolower`	`conv`(3C)	convert character to lowercase
`toupper`	`conv`(3C)	convert character to uppercase

Time functions

These functions are used for gaining access to and reformatting the current date and time values of the system. These functions are located and loaded automatically during the compiling of a C language program. No command-line request is needed because these functions are part of the C library.

You should include the header file associated with these functions near the beginning of any file using the time functions:

```
#include <time.h>
```

These functions (except `tzset`) convert a time such as returned by `time`(2):

Function	*Reference*	*Brief description*
`asctime`	`ctime`(3C)	return string representation of date and time
`ctime`	`ctime`(3C)	return string representation of date and time, given integer form
`gmtime`	`ctime`(3C)	return Greenwich mean time
`localtime`	`ctime`(3C)	return local time
`tzset`	`ctime`(3C)	set time-zone field from environment variable

Miscellaneous functions

These functions support a wide variety of operations:

- numeric conversion
- DES algorithm access
- group file access
- password file access
- parameter access
- hash table management
- binary tree management
- table management
- memory allocation
- pseudorandom number generation

These functions are automatically located and included in a program being compiled. No command-line request is needed because these functions are part of the C library.

Some of these functions require declarations to be included. These are described following the descriptions of the functions.

Numeric conversion

The following functions perform numeric conversion:

Function	*Reference*	*Brief description*
`a64l`	`a64l`(3C)	convert string to base 64 ASCII
`atof`	`atof`(3C)	convert string to floating
`atoi`	`atof`(3C)	convert string to integer
`atol`	`atof`(3C)	convert string to `long`
`frexp`	`frexp`(3C)	split floating into mantissa and exponent
`l3tol`	`l3tol`(3C)	convert 3-byte integer to `long`

`ltol3`	`l3tol`(3C)	convert `long` to 3-byte integer
`ldexp`	`frexp`(3C)	combine mantissa and exponent
`l64a`	`a64l`(3C)	convert base 64 ASCII to string
`modf`	`frexp`(3C)	split mantissa into integer and fraction

DES algorithm access

The following functions allow access to the Data Encryption Standard (DES) algorithm used on the A/UX operating system. (Not present in international distributions.) The DES algorithm is implemented with variations to frustrate use of hardware implementations of the DES for key search:

Function	*Reference*	*Brief description*
`crypt`	`crypt`(3C)	encode string
`encrypt`	`crypt`(3C)	encode/decode string of 0's and 1's
`setkey`	`crypt`(3C)	initialize for subsequent use of `encrypt`

Group file access

The following functions are used to obtain entries from the group file (stored in `/etc/group`). You must include declarations for these functions in the program being compiled with the line

```
#include <grp.h>
```

Function	*Reference*	*Brief description*
`endgrent`	`getgrent`(3C)	close group file being processed
`getgrent`	`getgrent`(3C)	get next group file entry
`getgrgid`	`getgrent`(3C)	return next group with matching group ID
`getgrnam`	`getgrent`(3C)	return next group with matching name
`setgrent`	`getgrent`(3C)	rewind group file being processed
`fgetgrent`	`getgrent`(3C)	get next group file entry from a specified file

Password file access

These functions are used to search for and gain access to information stored in the password file (`/etc/passwd`). Some functions require declarations that you can include in the program being compiled by adding the line

```
#include <pwd.h>
```

Function	*Reference*	*Brief description*
`endpwent`	`getpwent`(3C)	close password file being processed
`getpw`	`getpw`(3C)	search password file for user ID
`getpwent`	`getpwent`(3C)	get next password file entry
`getpwnam`	`getpwent`(3C)	return next entry with matching name
`getpwuid`	`getpwent`(3C)	return next entry with matching user ID
`putpwent`	`putpwent`(3C)	write entry on stream
`setpwent`	`getpwent`(3C)	rewind password file being examined
`fgetpwent`	`getpwent`(3C)	get next password file entry from specified file

Parameter access

The following functions provide access to several different types of parameters. None requires any declarations:

Function	*Reference*	*Brief description*
`getopt`	`getopt`(3C)	get next option from option list
`getcwd`	`getcwd`(3C)	return string representation of current working directory
`getenv`	`getenv`(3C)	return string value associated with environment variable
`getpass`	`getpass`(3C)	read string from terminal without echoing
`putenv`	`putenv`(3C)	change or add value of environment variable

Hash table management

The following functions are used to manage hash search tables. You should include the header file associated with these functions in the program being compiled. You can do so by including the line

```
#include <search.h>
```

near the beginning of any file using the search functions:

Function	*Reference*	*Brief description*
`hcreate`	`hsearch`(3C)	create hash table
`hdestroy`	`hsearch`(3C)	destroy hash table
`hsearch`	`hsearch`(3C)	search hash table for entry

Binary tree management

These functions are used to manage a binary tree. You should include the header file associated with these functions near the beginning of any file using the search functions:

```
#include <search.h>
```

Function	*Reference*	*Brief description*
`tdelete`	`tsearch`(3C)	delete nodes from binary tree
`tfind`	`tsearch`(3C)	find element in binary tree
`tsearch`	`tsearch`(3C)	look for and add element to binary tree
`twalk`	`tsearch`(3C)	walk binary tree

Table management

These functions are used to manage a table. Because none of these functions allocate storage, sufficient memory must be allocated before using these functions. You should include the header file associated with these functions near the beginning of any file using the search functions:

```
#include <search.h>
```

Function	*Reference*	*Brief description*
`bsearch`	`bsearch`(3C)	search table using binary search
`lsearch`	`lsearch`(3C)	look for and add element in table (linear search)
`lfind`	`lsearch`(3C)	find element in table (linear search)
`qsort`	`qsort`(3C)	sort table using quick-sort algorithm

Memory allocation

To use these routines, either include the following line in your program:

```
#include <malloc.h>
```

or compile your program with the command

`cc [`*option* ...`] [`*file* ...`] -lmalloc`

or both.

The following functions provide a means by which memory can be dynamically allocated or freed:

Function	*Reference*	*Brief description*
`calloc`	`malloc`(3C)	allocate zeroed storage
`free`	`malloc`(3C)	free previously allocated storage
`malloc`	`malloc`(3C)	allocate storage
`realloc`	`malloc`(3C)	change size of allocated storage

The following is another set of memory allocation functions available. They are faster than the (3C) versions, but require more memory.

Function	*Reference*	*Brief description*
`calloc`	`malloc`(3X)	allocate zeroed storage
`free`	`malloc`(3X)	free previously allocated storage
`malloc`	`malloc`(3X)	allocate storage
`mallopt`	`malloc`(3X)	control allocation algorithm
`malinfo`	`malloc`(3X)	space usage
`realloc`	`malloc`(3X)	change size of allocated storage

Pseudorandom number generation

The following functions are used to generate pseudorandom numbers. The function names that end with 48 are a family of interfaces to a pseudorandom number generator based on the linear congruent algorithm and 48-bit integer arithmetic. The `rand` and `srand` functions provide an interface to a multiplicative congruent random number generator with a period of 232.

◆ **Note** For intervals, the notation [*a* to *b*] means that *a* and *b* are included in the range, whereas the notation (*a* to *b*) means that *a* and *b* are not included, but all points in between are in the range. Therefore, the notation [*a* to *b*) means that *a* is included, as is everything from *a* to *b*, and *b* is not included. ◆

Function	*Reference*	*Brief description*
`drand48`	`drand48`(3C)	random `double` over the interval [0 to 1)
`lcong48`	`drand48`(3C)	set parameters for `drand48`, `lrand48`, and `mrand48`
`lrand48`	`drand48`(3C)	random `long` over the interval [0 to 2^{31})
`mrand48`	`drand48`(3C)	random `long` over the interval [– 2^{31} to 2^{31})
`rand`	`rand`(3C)	random integer over the interval [0 to 32767)
`seed48`	`drand48`(3C)	seed the generator for `drand48`, `lrand48`, and `mrand48`

Function	*Reference*	*Brief description*
`srand`	`rand`(3C)	seed the generator for `rand`
`srand48`	`drand48`(3C)	seed the generator for `drand48` `lrand48`, and `mranb48` using a `long`

Signal handling functions

The functions `gsignal` and `ssignal` implement a software facility similar to `signal`(3) in *A/UX Programmer's Reference*. This facility lets you indicate the disposition of error conditions and allows you to handle signals for your own purposes. The declarations associated with these functions should be included near the beginning of any file using the signal handling functions.

```
#include <signal.h>
```

These declarations define ASCII names for the 15 software signals.

Function	*Reference*	*Brief description*
`gsignal`	`ssignal`(3C)	send a software signal
`ssignal`	`ssignal`(3C)	arrange for handling of software signals

Miscellaneous

These functions do not fall into any previously described category:

Function	*Reference*	*Brief description*
`abort`	`abort`(3C)	cause an IOT signal to be sent to the process
`abs`	`abs`(3C)	return the absolute integer value
`ecvt`	`ecvt`(3C)	convert `double` to string
`fcvt`	`ecvt`(3C)	convert `double` to string using Fortran format
`gcvt`	`ecvt`(3C)	convert `double` to string using Fortran F or E format
`isatty`	`ttyname`(3C)	test whether integer file descriptor is associated with a terminal

`mktemp`	`mktemp`(3C)	create filename using template
`monitor`	`monitor`(3C)	cause process to record a histogram of program counter location
`swab`	`swab`(3C)	swap and copy bytes
`ttyname`	`ttyname`(3C)	return pathname of terminal associated with integer file descriptor

6 C Special Libraries

A/UX provides two special C libraries, the math library and the object-file library. This chapter describes both of these libraries.

As stated in Chapter 5, a library is a collection of related functions and declarations. All the functions described here are also described in Section 3 of *A/UX Programmer's Reference*. Most of the declarations described in this chapter can be found in `math`(5) in *A/UX Programmer's Reference*.

Introduction to the C math library

The C math library is made up of functions and a header file. The functions can be located and loaded during compile time if you make this request on the command line:

```
cc file.c -lm
```

This causes the link editor to search the math library. In addition to the request to load the functions, you should include the header file of the math library near the beginning of the first file being compiled:

```
#include <math.h>
```

The math library functions

The math-library functions are grouped into the following categories:

- trigonometric functions
- Bessel functions
- hyperbolic functions
- miscellaneous functions

Trigonometric functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double precision.

Function	*Reference*	*Brief description*
`acos`	`trig`(3M)	return arc cosine
`asin`	`trig`(3M)	return arc sine
`atan`	`trig`(3M)	return arc tangent
`atan2`	`trig`(3M)	return arc tangent of a ratio
`cos`	`trig`(3M)	return cosine
`sin`	`trig`(3M)	return sine
`tan`	`trig`(3M)	return tangent

Bessel functions

These functions calculate Bessel functions of the first and second kinds of several orders for real values. `j0`, `j1`, and `jn` are Bessel functions of *x* of the first kind, while `y0`, `y1`, and `yn` are Bessel functions of *x* of the second kind. The value of *x* must be positive:

Function	*Reference*	*Brief description*
`j0`	`bessel`(3M)	give result of order 0
`j1`	`bessel`(3M)	give result of order 1
`jn`	`bessel`(3M)	give result of order *n*
`y0`	`bessel`(3M)	give result of order 0
`y1`	`bessel`(3M)	give result of order 1
`yn`	`bessel`(3M)	give result of order *n*

Hyperbolic functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values:

Function	*Reference*	*Brief description*
`cosh`	`sinh`(3M)	return hyperbolic cosine
`sinh`	`sinh`(3M)	return hyperbolic sine
`tanh`	`sinh`(3M)	return hyperbolic tangent

Miscellaneous functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double-precision numbers:

Function	*Reference*	*Brief description*
`ceil`	`floor`(3M)	return the smallest integer not less than a given value
`exp`	`exp`(3M)	return the exponential function of a given value
`fabs`	`floor`(3M)	return the absolute value of a given value

Function	*Reference*	*Brief description*
`floor`	`floor`(3M)	return the largest integer not greater than a given value
`fmod`	`floor`(3M)	return the remainder produced by the division of two given values
`gamma`	`gamma`(3M)	return the natural log of the absolute value of the result of applying the gamma function to a given value
`hypot`	`hypot`(3M)	return the square root of the sum of the squares of two numbers
`log`	`exp`(3M)	return the natural logarithm of a given value
`log10`	`exp`(3M)	return the logarithm base ten of a given value
`matherr`	`matherr`(3M)	error-handling function
`pow`	`exp`(3M)	return the result of a given value raised to another given value
`sqrt`	`exp`(3M)	return the square root of a given value

Introduction to the C object-file library

The C object-file library provides functions for accessing and manipulating object files. Some of these functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. For a description of object-file format, see Chapter 16, "COFF Reference," in this manual.

These functions are usually used only by compilers, link editors, cross-reference generators, and so on. Most applications programmers do not need to use them.

The object-file library functions reside in `/usr/lib/libld.a` and can be located and loaded at compile time if you give the following command-line request:

`cc` *file* `-lld`

The command causes the link editor to search the object-file library. The argument `-lld` must appear after all files that reference functions in `libld.a`.

In addition, you must include various header files:

```
#include <stdio.h>

#include <a.out.h>

#include <ldfcn.h>
```

The object-file library functions

Function	*Reference*	*Brief description*
`ldaclose`	`ldclose`(3X)	close object file
`ldahread`	`ldahread`(3X)	read archive header
`ldaopen`	`ldopen`(3X)	open object file for reading
`ldclose`	`ldclose`(3X)	close object file being processed
`ldfhread`	`ldfhread`(3X)	read file header of object file being processed
`ldgetname`	`ldgetname`(3X)	retrieve name of an object-file symbol table entry
`ldlinit`	`ldlread`(3X)	prepare object file for reading line number entries through `ldlitem`
`ldlitem`	`ldlread`(3X)	read line number entry from object file after `ldlinit`
`ldlread`	`ldlread`(3X)	read line number entry from object file
`ldlseek`	`ldlseek`(3X)	seek to the line number entries of the object file being processed
`ldnlseek`	`ldlseek`(3X)	seek to the line number entries of the object file being processed given the name of a section
`ldnrseek`	`ldrseek`(3X)	seek to the relocation entries of the object file given the name of a section
`ldnshread`	`ldshread`(3X)	read section header of the named section of the object file
`ldnsseek`	`ldsseek`(3X)	seek to the section of the object file being processed given the name of a section
`ldohseek`	`ldohseek`(3X)	seek to the optional file header of the object file being processed

Function	*Reference*	*Brief description*
`ldopen`	`ldopen`(3X)	open object file for reading
`ldrseek`	`ldrseek`(3X)	seek to the relocation entries of the object file being processed
`ldshread`	`ldshread`(3X)	read section header of an object file being processed
`ldsseek`	`ldsseek`(3X)	seek to the section of the object file being processed
`ldtbindex`	`ldtbindex`(3X)	return the long index of the symbol table entry at the current position of the object file being processed
`ldtbread`	`ldtbread`(3X)	read a specific symbol table entry of the object file being processed
`ldtbseek`	`ldtbseek`(3X)	seek to the symbol table of the object file being processed
`sgetl`	`sputl`(3X)	access long integer data in a machine-independent format
`sputl`	`sputl`(3X)	translate a long integer into a machine-independent format

Common object-file interface macros (`ldfcn.h`)

The interface between the calling program and the object-file access routines is based on the defined type `ldfile`, which is defined in the header file `ldfcn.h` (see `ldfcn`(3X)). The primary purpose of this structure is to provide uniform access both to simple object files and to object files that are members of an archive file.

The function `ldopen` allocates and initializes the `ldfile` structure and returns a pointer to that structure to the calling program. You can gain access to the fields of the `ldfile` structure individually through the following macros.

Macro	*Reference*	*Brief description*
`type`	`ldfcn`(3X)	return the magic number of the file, which is used to distinguish between archive files and simple object files
`IOPTR`	`ldfcn`(3X)	return the file pointer that was opened by `ldopen` and is used by the input/output functions of the C library
`OFFSET`	`ldfcn`(3X)	return the file address of the beginning of the object file; this value is nonzero only if the object file is a member of the archive file
`HEADER`	`ldfcn`(3X)	access the file header structure of the object file

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an `ldfile` structure into a reference to its file descriptor field. The available macros are described in `ldfcn`(3X) in *A/UX Programmer's Reference.*

7 Shared Libraries

A shared library is similar in function to a normal, nonshared library. For the developer compiling a program, specifying a shared library on the command line is done just as with a nonshared library. There is no functional difference for the application user who invokes the resulting application, except that the application using a shared library can yield certain efficiency benefits.

This chapter is presented in two parts. The first part, "Using a Shared Library," explains what a shared library is and the benefits you can obtain by using a shared-library version rather than a nonshared-library version of an archive.

The second section of this chapter, "Building a Shared Library," provides information to library developers and advanced programmers who are building shared libraries. You do not need to read this section to use shared libraries. For library developers, this section describes how to prepare a specification file for a shared library and how to use `mkshlib`(1) to create a shared library from that specification file and the object files specified on the command line. An example specification file is provided.

Using a shared library

This section describes what a shared library is and how to use one to build executable object files. The section also describes the benefits and drawbacks of using a shared library. Finally, it tells how to determine whether an executable object file uses a shared library.

What is a shared library?

When a program calls on a shared library for library routines, those routines are made available at run time to that program, and to any other program that calls on that shared library and happens to be running at the same time. Each program receives its own copy of the `.data` segment portion of the shared-library routines, but all programs share the `.text` segment of the shared library. (In contrast to the shared library, each program that makes use of a nonshared library receives a private copy of any library routines required.)

A shared library actually consists of two files (two sublibraries) containing source archives and executable object files, referred to as the host file and the target file, respectively. The executable code for a shared library is in the Common Object File Format (COFF). This code is accessed from the applications that call it by means of a special addressing structure provided within the application during link-edit.

The host and target files can be on different systems. A host file is an archive that provides information used during link-edit. (Chapter 15 of this manual provides information about the link editor, `ld`. For additional information about archive libraries, see `ar`(4).) The name of the host file is included on the compilation command line in the same way as with a nonshared library. All operations that can be performed on a nonshared library can be performed on a host file.

The target file contains the executable code for all the routines in the library. This library is brought into memory, if not already present, during execution of a program that calls upon it. The library is attached to a user's process during execution.

How do shared libraries work?

Shared libraries are built using the process described in "Building a Shared Library," later in this chapter. The `mkshlib` utility uses information given to it in a specification file to construct a host and a target file. The specification file names the object files from which a shared library can be constructed. To oversimplify, the process involves splitting up sections from these object files. The target file receives the `.text`, `.data`, and `.bss` sections.

The host file has symbol information, used by the link editor, for all sections in both libraries.

When a compilation command specifies a host file, the executable object file that results receives a special section called `.lib`, which contains a pathname to the target shared library.

At execution time, the first invocation of a target shared library by an application results in the calling application being linked to the `.text`, `.data`, and `.bss` sections of that library. Successive applications referencing that target shared library while it is in memory link to the `.text` section and to private copies of the `.data` and `.bss` sections.

Invoking a shared library

Link editing or compiling with a shared library is done in the same way as with a nonshared library. The name of the host file is supplied on the command line. Shared-library files have a `_s` suffix to distinguish the shared-library version from the nonshared-library version. For example, `libc_s` is the shared-library version of `libc`, the standard C library. Here is the pattern for the `cc` command line.

`cc` *source-file* `-l`*host-library-file*

For the shared-library version of the standard C library, `libc_s`, the host filename is `c_s`, as shown here:

`cc` *file*`.c -lc_s`

Here is an example using that host:

```
cc  hello_world.c  -lc_s
```

The relocatable (nonshared) C library is still available; this library is searched by default during the compilation or link editing of C programs.

To link all the files in your current directory with `libc_s`, use the following command:

```
cc *.c -lc_s
```

The search for symbol definitions proceeds from one library archive to another in the order they are specified on the command line, until the first definition is found. Normally, you should include the `-lc_s` argument after all other `-l` arguments on the command line. The shared C library is treated like the relocatable C library, which is searched, by default, after all other libraries specified on a command line are searched. (If the argument for the standard C library, `-lc`, is on the command line, `-lc_s` must precede it. Otherwise, the standard library is used before the shared version can be invoked.)

You should not have to change the code in any applications you already have when you use a shared library with them.

Application source code in C or assembly language is compatible with both nonshared- and shared-library archives. When coding a new application for use with a shared library, you should use your standard coding conventions.

Benefits of using a shared library

A shared library offers several benefits for individual users and for the system as a whole. For each application that calls on a shared library rather than a nonshared library, the application can gain these benefits:

Disk storage space savings	Because shared-library code is not copied in all the executable object files that use the code, these files are smaller and use less disk space.
Memory savings	Because they share library code at run time, the dynamic memory needs of the process are reduced.

Easier executable file maintenance

Updating a shared library effectively updates all executable files using that library. Correcting an error in shared-library code, or enhancing that code, provides the benefits of the new code to all processes that use the library.

In contrast, a nonshared library cannot provide this maintenance benefit. Changes to their archive libraries do not affect executable files made earlier, because code is copied to the files during link editing rather than during execution.

These individual benefits accrue to the system. Savings in storage space for many individual applications are multiplied for general storage savings. Smaller processes provide efficiencies when swapping applications.

The capability for current maintenance is an important user benefit. For development work, using a shared library ensures that team members are using the most current routines. The benefits are similar for application work, in which using a shared library can ensure that all data is processed using the same version of the required routines.

The A/UX shared-library directory

A/UX currently provides two target files (`libc_s` and `libcmac_s`), both in the directory `/shlib`, which is the suggested location for target files. As other shared libraries become available from software vendors or from your own development, they should be placed in that directory.

The `_s` suffix is a convention used to distinguish the shared-library version from the nonshared-library version. For example, `libc_s` is the shared-library version of `libc`, the standard C library. `libcmac_s` is the shared version of `libcmac`, the glue routines that access the Macintosh Toolbox. The host library for `lib_s` is `libc_s.a` and is located in the `/lib` directory. The host file for `libcmac_s` is `libcmac_s.a` and is located in the `/usr/lib` directory.

Space savings from using a shared library

A well-designed shared library almost always saves space. To determine what savings are gained from using a shared library, you can try building the same application with both a nonshared and shared library, assuming both versions are available. (Source code is compatible with either form of library.) Then, compare the two versions of the application for size and performance. Here is a demonstration you can enter and try immediately:

```
% cat hello.c
main()
{
     printf("Hello world\n");
}
$ cc -o unshared hello.c
$ cc -o shared hello.c -lc s
$ ls -l unshared shared
-rwxrwxrwx  2  jim  12658  Nov 11  unshared
-rwxrwxrwx  2  jim   7980  Nov 11  shared
```

The `ls -l` command shows the actual size of the object files. In this example, the sizes are 12658 bytes and 7980 bytes for the unshared and shared-library options. The `size`(1) command is not accurate for this purpose.

Archive library cautions

Here are some points to keep in mind when using a nonshared or shared archive library:

- Don't define symbols in your application with the same names as those in a library.
- Although there are exceptions, avoid redefining standard library routines, such as `printf` and `strcmp`. Replacements that are incompatibly defined can cause any library, shared or not, to behave incorrectly.
- Don't use undocumented library routines.
- Don't try to manipulate the underlying implementation, which is subject to change.

How using a shared library can increase space usage

A host file can add space to an executable object file, if the library has unresolved references. The `ld` link editor uses static linking, which requires that all external references in a program be resolved before the program is executed. A shared library can have imported symbols, which are used but not defined by the library. These symbols can introduce unresolved references during the linking process. To resolve these references, the link editor must add the `.init` section of the corresponding routine (from the host file) to the `.text` section of the executable object file, which increases the size of the executable object file.

A target file can increase the memory requirements of a process. Again, recall from "How Do Shared Libraries Work?" in this chapter that a shared-library target file can have both text and data regions connected to a process. Although the text region is shared by all processes that use the library, the data region is not. Each process using the shared library receives its own copy of the entire data region. Naturally, this region adds to the memory requirements of the process. If an application uses only a small part of a shared library text and data, then executing the application can require more memory with a shared library than without it.

For example, it is unwise to use the shared C library to access only `strcmp`. Compiling with a nonshared version of the library places only `strcmp` in the executable object file. Compiling with the shared version, while producing a slightly smaller version of the executable object file, means that a private copy of the `.data` and `.bss` sections of the target shared library is placed in storage, reserved for the executable object file. The memory cost outweighs the savings. The nonshared-library version is more appropriate.

When not to use a shared library

There are various situations for which the use of a shared library is not recommended. The previous section "How Using a Shared Library Can Increase Space Usage" points out some of them. Some other cases are listed here.

When making your decision about which form of library to use, remember that shared libraries are not available on versions of A/UX prior to Release 2.0. If your application must run on prior versions, you must use a nonshared library.

During debugging, you might need to use a nonshared-library version if you encounter certain difficulties. See "Debugging Files That Use Shared Libraries," later in this chapter, for more information.

Identifying files that use shared libraries

To determine whether an executable file uses a shared library, you can use the `dump(1)` command to look at the section headers for the file.

If the file has a `.lib` section, a shared library is needed. If the file has no `.lib` section, it does not use a shared library. The command to use is

```
dump -hv filename
```

If the file uses a shared library, the display also shows sections corresponding to target file sections. These are dummy sections and do not contain actual section data.

Debugging files that use shared libraries

Debugging support for shared libraries is currently limited. Information from shared libraries is not dumped to core files and `sdb(1)` does not read the symbol tables of shared libraries. You can use `sdb` to single step through shared-library code, but cannot set breakpoints in the shared-library area. If you encounter an error that appears not to be in your application code, you might find debugging easier if you recompile the application with the nonshared version of the library. See Chapter 9 for more information on `sdb`.

Building a shared library

This section describes the process of building shared libraries in several phases leading up to the execution of the `mkshlib` command that is used to build and maintain shared libraries. The first phase is designing the shared library, in which the routines or object files to go into the library are selected. The next phase is preparing the object files that are to go into the shared library. The third phase is to prepare the specification file that describes the shared library to the `mkshlib` command. The final phase is executing the `mkshlib` command to build the host and target files, which can be done at one time or with separate invocations of the `mkshlib` command.

In practice, the entire sequence of phases can be done iteratively. In developing a production shared library, you can build several versions of a shared library, with more or fewer object files, and try them in practical use to determine the best combination, rather than attempting to settle all design and selection questions before proceeding.

Also, once the preparatory work is done and a specification library is available, the `mkshlib` command can be executed to make new copies of the host or target library. Minor changes can be made to the specification file, such as changing the target library pathname.

Designing a shared library

This phase consists of selecting what is appropriate to put in a shared library. Routines that have little code in comparison to their `.data` and `.bss` sections are not good candidates, because only the code portion can actually be shared.

The routines to be included should be often-used routines. The entire target library is brought into memory to serve applications that call upon it. When that target library includes routines seldom used by the applications calling on it, then space is wasted. Less-used routines should be made available in nonshared form, where they are included in only those applications that actually use them.

You might wish to develop more than one shared library customized for groups that need a particular combination of routines, rather than including a variety of routines in one general shared library.

When a shared library is used during software development, considerations of space or frequency of use can be overridden by the importance of having certain routines in common use by all software modules under development.

Building different versions of a shared library and profiling actual use can be used to settle certain design questions.

One feature of shared libraries is that the host file can contain both nonshared routines and linking information for shared routines. Such a host file allows sharing of often-used routines with access to less-often-used routines. When the application file is linked to that host file, any nonshared routines that are referenced are copied in as usual; shared routines are accessed at time of execution.

Such a host library is developed as follows. The host is built using `mkshlib` and contains as shared-library routines those files listed in the specification file under the `#objects` directive. The nonshared routines are added afterward, using the archiver program (see `ar`(1)). One of the host files built is `/lib/libc_s.a`.

Handling external references

If desired, a shared library can reference routines and variables not contained within the target file. These are called external references.

If you have no external references, the information that follows does not apply. You can proceed to "Preparing a Shared Library," later in this chapter.

All external references must be resolved, which is done in two phases of the development process: when preparing object files for inclusion in the library and when developing the specification file.

Prepare an include file that aliases all imported variables (variables external to a routine, the value of which must be imported). All such references, whether to a routine or to a variable, are defined by specifying a pointer, in the form

`#define` *import pointer*

When compiling the object files, include this file in every source file that requires it to resolve imported variables.

Next, create a source file with declarations to initialize all the imported variables to zero or null. Use statements of the form

```
int (*pointer) = 0;
```

The type specified on the left must match the type required for the variable. Compile this source to produce an object file and include this object file in the object file specification list, preferably as the first file. In the example specification file, this file is `def.o`.

When preparing the specification file, provide an initialization line for every external reference.

What is the effect of all this? Using the include file and the declaration file described here provides resolution of external references that allows a self-contained target file to be built. The target file code contains null pointers for all these references, but the information necessary to provide the true values is available in the `.data` section.

The initialization line in the specification file informs the `mkshlib` command that initialization code is required. The code is then developed, using information on where the required value can be obtained. The initialization code goes into a special section named `.init`, which is placed in the host file. Each object file requiring initialization has an `.init` section.

When an application linked to a host file uses an object file that has an associated `.init` section, then a copy of the required `.init` section is placed in the application executable file.

Preparing a shared library

The object files selected for inclusion should be compiled without the `-g` flag (debug) option. If this rule is violated, the process of building a shared library fails.

All data files, global and static, should be listed under the `#objects` directive in the specification file. It is good practice to place global data and static data in separate files. Interspersing global data, static data, and regular objects in one file can lead to unexpected behavior when using the library. In the example shown under "Specification File Example," later in this chapter, the global data file is `def.o`.

The files to be included do not usually need any special reworking. If files contain external references, see "Handling External References," earlier in this chapter.

The `mkshlib` command

The `mkshlib` command is used to build and maintain shared libraries. The command can be used to build both host and target libraries or only one of these. The `mkshlib` command requires the name of a specification file that contains information necessary to build the host and target files.

The user interface to `mkshlib` consists of this information and command-line arguments:

`mkshlib` *specs* `[ -n ] -t` *target* `[ -h` *host* `]`

To build both files, provide both names. For example,

```
mkshlib -s myspec -t lib_s -h lib_s.a
```

To build only the target file, do not provide a host name. For example,

```
mkshlib -s myspec -t lib_s
```

A host file is required to access the target file through the link-edit process. In the example above, the host file can be on a different system and the command can be building a local target file, `lib_s`. The specification file `myspec` establishes a pathname to the target file.

The `-n` option can be used to build only a new host file. For example,

```
mkshlib -s myspec -t lib_s -h lib_s.a -n
```

The name of a target file must be supplied, although only the host is to be built. In the example the target file name is `lib_s`.

To build the host and target files, `mkshlib` invokes other tools, such as the archiver, `ar`(1), the assembler, `as`(1), and the link editor, `ld`(1).

Command-line arguments

The following command-line arguments are recognized:

`-s` *specs*	Provide the name of the shared-library specification file, *specs,* which contains the information necessary to build the shared library. Its contents include a list of the object files to be included in the shared library, the branch-table specifications for the target file, the pathname where the target file is to be created, and the start addresses of the `.text` and `.data` sections for the target file. Initialization specifications for imported variables are given in this file, if necessary. Imported variables are addresses external to the target file, such as the addresses of routines and variables that the library can call upon. Details about the shared-library specification file are given in the next section, "The Shared-Library Specification File."
`-t` *target*	Specify the name, *target,* of the target file to be produced. The location where the target file is to be built can be different from the location specified in the `#target` directive of the specification file. However, the target file can function only when placed in the location given in the specification file, with execution permission set.
`-h` *host*	Specify the name of the host file, *host.* If not specified, then the host file is not produced. The host file can be built in a convenient directory and moved later to the appropriate directory (`/lib` or `/usr/lib`).
`-n`	Do not generate a new target file. This option is used to update the host file only. The `-t` flag option and the target filename must still be supplied, because a version of the target file is needed to build the host file.

The shared-library specification file

The specification file contains all the information necessary to build both the host and target shared libraries. The file contains directive names and associated specification information. Directive names must be at the start of the line. Some directives have specification information on the same line, and some directives introduce multiple specifications on following lines. Lines following such a directive are interpreted as specification lines for that directive, until another directive or the end of the file is encountered.

Specification file structure

The six possible directives are the following:

`##` *comment-text*

`#address` *section-name address*

`#branch`

`#init` *object*

`#objects` *file*

`#target` *pathname*

Their use is described in the following paragraphs. Directives can be given in any order in the specification file, except for the `#init` directive.

`##` *comment-text*
Specifies that the remainder of the line is a comment. All *comment-text* on that line is ignored. Comment lines can occur anywhere. Comments are optional, but recommended.

`#address` *section-name address*
Specify the start *address* in the virtual address space at which to bind the *section-name* of the target file. Typically, address directives are provided for the `.text` and `.data` sections of the target file. Addresses must be on a 256 kilobyte (KB) boundary, which corresponds to the current memory management segment size.

The `.bss` section is grouped with the `.data` section and does not require a start address.

There are constraints on the choice of addresses. The address cannot be the same as those specified for any other shared library, unless the two target shared libraries are never used at the same time. The address specifications for the two target shared libraries currently provided with A/UX are as follows:

```
libc_s      .text       0x47f00000
            .data       0x47fc0000

libmac_s    .text       0x47e00000
            .data       0x47ec0000
```

The address values specified using the `#address` directive should be in the range 48000000 through 50000000. The addresses starting at 40000000 and ending below 48000000 are reserved.

`#branch`

The branch-table specifications appear in the following format:

branch-table-specification
branch-table-specification
branch-table-specification

.
.
.

All lines following the `#branch` directive are interpreted as branch-table specifications, until another directive is encountered. Only one `#branch` directive can be in a specification file. The branch table built from these specifications consists of jump instructions to the specified functions.

Branch-table specification lines have the following format:

function-name position

Only functions should be given branch-table entries, and those functions must be external. Each *function-name* can appear only once. The position value is the slot location of the function name in the branch table. The value of *position* for each *function-name* given is the slot (or range of slots taken). The value of *position* is a single integer or a range of integers of the form *position1 - position2*. (The use of a position range is given later.) Values start with 1, each position value can be used only once, and all position values from 1 to the highest value used must be accounted for.

A position range also can be used to reserve empty slots in the branch table for later use. Only the highest value of the range is associated with the function name. The remaining positions in the range can be used later for other functions.

When adding functions to an existing library, provide the new functions at higher positions than in the existing branch table. Changing positions in an existing branch table renders that shared library unusable by previously linked applications.

`#init` *object*

initialization

initialization

initialization

.

.

.

Specify *object* with the name of an object file that requires initialization code because it uses an imported variable. Each object file that requires initialization must be specified. If the shared library being built is completely self-contained (uses no imported variable), then no `#init` directive is used because no initialization code is necessary.

All `#init` directives must be placed after the `#objects` directive and its associated specifications in the specification file.

An `#init` directive is followed by one or more *initialization* specification lines pertaining to the object file, *object*, named in the directive. Each line following the directive is interpreted as a specification line until another directive is encountered. Specify each line of *initialization* by using the following format:

import pointer

The placeholder *import* refers to an imported variable, and *pointer* is a pointer defined within the object file named in the `#init` directive preceding the initialization line. For each initialization line so specified, initialization code is generated in the form

```
pointer = &import;
```

in which the value of *pointer* is set to the absolute address of `import`. This initialization code is placed in the corresponding object file in the host file.

For additional information, see "Handling External References," earlier in this chapter.

```
#objects file
file
file
file
  .
  .
  .
```

Specify each entry of *file* with the names of the object files constituting the target shared library.

This directive can be specified only once per shared-library specification file. The lines following the directive are interpreted as specifications of *file* until another directive is encountered.

`#target` *pathname*
Specify the absolute path for the location of the target file on the target system. This pathname is copied into executable object files and tells the operating system where to find the target file when executing a file that uses it. The maximum length for *pathname* is 64 characters.

Specification file example

The specification file specifies controlling information to `mkshlib` about how the shared library is to be developed. "The Shared-Library Specification File," earlier in this chapter, gives detailed information about the statements that are used in the specification file.

The following example shows how specification statements work together. There are six types of statements: a comment statement, which `mkshlib` ignores, and five that are interpreted by `mkshlib`. The example that follows shows all six types:

```
##  Example Shared Library

#target /shlib/example_s


#address  .text  0x47f00000
```

```
#address  .data  0x47fc0000


## Only one branch table allowed.
#branch
malloc   1
free     2
realloc  3
sbrk     4
cerror%  6
memcpy   7


#objects
         def.o
         extdata.o
         malloc.o
         sfiles/sbrk.o
         sfiles/cerror.o


## Init statement(s) must be after #objects statements
#init  def.o
         end     libc_end
```

An explanation of the example file follows. First, look at the general layout of the example. Notice that blank lines can be inserted for readability. A comment line tells about the file.

```
##  Example Shared Library
```

The `##` shows that this is a comment line. The `mkshlib` utility ignores this line when using the specification file. As the rest of the example shows, comment lines can occur between any other lines without affecting the interpretation of this file.

The `#target` statement establishes the pathname where the target file is read or created.

```
#target  /shlib/example_s
```

The two `#address` statements provide the locations for the `.text` and `.data` segments of the target file when it is brought into memory. The target file is one file.

```
#address  .text  0x47f00000

#address  .data  0x47fc0000
```

The `#branch` statement signals the start of the branch table.

```
#branch
malloc   1
free     2
realloc  3
sbrk     4
cerror%  6
memcpy   7
```

The branch table lists the names of all functions in the library that are available externally. The branch table that is constructed from this specification contains a jump statement to each named routine. Any function within the library that is not called from outside the library does not need to be listed. The numbers after the names are position numbers, specifying the slot in the branch table in which to place the jump statement.

The `#objects` statement introduces a list of object files in the library. This tells the `mkshlib` command what object files to process to produce the host and target file.

```
#objects
         def.o
         xtdata.o
         malloc.o
         sfiles/sbrk.o
         sfiles/cerror.o
```

The `#init` statement is required for this library because there is an unresolved reference in an object file, in this case, the `def.o` object file.

```
#init  def.o
         end     libc_end
```

The `#init` statement follows the `#objects` statement because the `def.o` object must be defined (listed under an `#object statement`) before the `#init` statement that refers to it. The line that follows the `#init` statement is called an initialization line.

```
end   libc_end
```

This initialization line assigns an address to `end`, the absolute address of `libc_end`. There is only one initialization line for the `#init def.o` statement because there is only one unresolved reference in `def.o`. The statement causes an `.init` section for `def.o` to be placed in the host file. There are no other `#init` statements because no other objects have unresolved references. For information on the `#init` statement, see "Handling External References," earlier in this chapter.

In the preparation of the object files that go into this example, here is how this reference is resolved. A header file, `def.h`, contains this statement:

```
#define   end   (*libc_end)
```

Every source file in the library that references `end` includes `def.h`.

The pointer is initialized with the C statement, in `def.c`, by the declaration

```
int   (*libc_end)   =   0;
```

The result of compiling such external references is an object file, `def.o` in this example, that should be placed first in the object file list.

Directory and file information

The `mkshlib` command is in the directory `/usr/bin/mkshlib`. The suggested directories for shared libraries are as follows:

`/lib/*_s.a`

or

`/usr/lib/*_s.a`	host (archive) library
`/shlib/*_s`	target (executable) library

Additional information

Additional information relating to topics discussed here can be found in the command reference and programmer's reference documentation: `ar`(1), `as`(1), `cc`(1), `ld`(1), `a.out`(4), and `ar`(4).

8 `lint` Reference

The `lint` program can be used to detect bugs, obscurities, inconsistencies, and portability problems in C programs. It is generally more restrictive than the C compiler. Constructions that the C compiler accepts without complaint, `lint` considers wasteful or error prone. The `lint` program is also more rigid than the C compiler with regard to the C language type rules. Also, `lint` accepts multiple files and library specifications and checks them for consistency.

You can suppress some or all of the `lint` checking mechanisms if they aren't necessary for a given application.

Using `lint`

The `lint` command has the form

`lint [`*option* ...`]` *file* ... *library-descriptor* ...

where *options* are optional flags that control `lint` checking and messages, *files* are the files to be checked by `lint` (files containing C language programs must have a `.c` extension; this is mandatory for both `lint` and the C compiler), and *library-descriptors* are the names of the libraries to be used in checking the program.

The `lint` library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined in a library file but aren't used in a source file do not result in messages.

The `lint` program does not simulate a full library search algorithm and prints messages if the source files contain a redefinition of a library routine.

Options

When you use more than one option, you should combine them into a single argument, such as `-ab` or `-xha`.

The options that are currently supported by the `lint` program are as follows:

`-a`	Use this option to suppress messages concerning the assignment of `long` values to variables that are not `long`. This option is often useful because there are a number of legitimate reasons for assigning `long` values to type `int`.
`-b`	Use this option to suppress messages concerning `break` statements that are unreachable. For example, programs generated by `yacc` and `lex` (see *A/UX Programming Languages and Tools,* Volume 2, for information on these programs) can have hundreds of unreachable `break` statements. If the C compiler optimizer were used, these unreachable statements would be of little importance, but the resulting messages would clutter up the `lint` output. The `-b` option takes care of this problem.
`-c`	Use this option to treat casts as though they were assignments subject to warning messages. (The default is to pass all legal casts without comment, no matter how bizarre the type mixing seems.)

`-h`	Use this option only to suppress the use of heuristics. By default, heuristics are used to check for wasteful or error-prone constructions and to detect bugs. For example, by default, `lint` prints messages about variables declared in inner blocks whose names conflict with the names of variables declared in outer blocks. Though this construction is considered legal, it is bad programming style and frequently a bug.
`-ly`	Use this option to specify libraries you want to include and have checked by `lint`. The source code is tested for compatibility with these libraries. This is done by getting access to library description files whose names are constructed from the library arguments. These files must all begin with the comment `/* LINTLIBRARY */` This comment must then be followed by a series of dummy function definitions. The critical parts of these definitions are as follows: ▪ the declaration of the function return type ▪ whether the dummy function returns a value ▪ the number and types of arguments to the function. The `VARARGS` and `ARGSUSED` comments can be used to specify features of the library functions.
`-n`	Use this option to suppress checking for compatibility with either the standard or the portable `lint` library. In effect, this option suppresses all library checking.
`-p`	Use this option to check program portability to other dialects of C language. This option checks a file containing descriptions of standard library routines that are expected to be portable.
`-u`	Use this option to suppress messages concerning function and external variables that are used and not defined or defined and not used. For more information, refer to "Unused Variables and Functions" later in this chapter.
`-v`	Use this option to suppress messages concerning unused function arguments. For more information, refer to "Unused Variables and Functions" later in this chapter.
`-x`	This option suppresses messages about variables referenced by external declarations but never used.
`-o` *name*	Use this option to create a `lint` library from input files named `llib-l`*`name`*`.ln`.

The `-D`, `-U`, and `-I` flag options of `cpp`(1) are also recognized as separate arguments. By default, `lint` checks the programs you give it against a standard library file that contains descriptions of programs normally loaded when a C program is run. When the `-p` option is used, another file is checked that contains descriptions of the standard library routines expected to be portable across various machines. You can use the `-n` option to suppress all library checking.

Message categories

The following subsections describe the major categories of messages printed by `lint`.

Unused variables and functions

As sets of programs evolve and develop, variables and function arguments that were used previously cannot be used in subsequent versions. It's not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. Although these types of errors rarely cause working programs to fail, they are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions occasionally can serve to help discover bugs.

The `lint` program prints messages about variables and functions that are defined but not otherwise mentioned.

You can suppress messages regarding variables that are declared through explicit `extern` statements but are never referenced. The statement

```
extern double sin();
```

evokes no comment if `sin` is never used, providing the `-x` option is used.

◆ **Note** This agrees with the semantics of the C compiler. ◆

If these unused external declarations are of interest, you can use `lint` without the `-x` option.

In some programming styles, many functions are written with similar interfaces. Frequently, some of the arguments are unused in many of the calls. The `-v` option is available to suppress the printing of messages about unused arguments, including those arguments that are unused and declared as register arguments. This can prevent a waste of the register resources of the machine.

To suppress such messages for one function only, add the comment

```
/* ARGSUSED */
```

to the program before the function. Also, you can use the comment

```
/* VARARGS */
```

to suppress messages about variable numbers of arguments in calls to a function. If you want to check the first several arguments and leave the later ones unchecked, include a digit giving the number of arguments that should be checked. For example,

```
/* VARARGS2 */
```

causes only the first two arguments to be checked.

One case in which information about unused or undefined variables is more distracting than helpful is when `lint` is applied to some but not all files out of a collection that is to be loaded at one time.

In this case, many of the functions and variables defined cannot be used. Conversely, many functions and variables defined elsewhere can be used. The `-u` option can be used to suppress the spurious messages that might otherwise appear.

Set/used information

The `lint` program attempts to detect cases where a variable is used before it is set. The `lint` program detects local variables (automatic and register storage classes) whose first use appears earlier than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," as the actual use can occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement because the true flow of control need not be discovered. It does mean that `lint` can print messages about some programs that are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The `lint` program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These are a frequent source of inefficiency and also can be symptomatic of bugs.

Flow of control

The `lint` program tries to detect unreachable portions of the programs that it processes. It prints messages about unlabeled statements immediately following `goto`, `break`, `continue`, or `return` statements. An attempt is made to detect loops that can never be left at the bottom and to recognize the special cases `while(1)` and `for(;;)` as infinite loops.

The `lint` program also prints messages about loops that cannot be entered at the top. Some valid programs can have such loops but they are considered to be bad style at best and bugs at worst.

The `lint` program has no way of detecting functions that are called and never returned. Thus, a call to `exit` can cause unreachable code that `lint` does not detect. This can seriously affect the determination of returned function values (see the next section, "Function Values"). If a particular place in the program cannot be reached but this is not apparent to `lint`, you can add the comment

```
/* NOTREACHED */
```

at the appropriate place. This informs `lint` that a portion of the program cannot be reached.

If you give the `-b` option, `lint` does not print a message about unreachable `break` statements. Programs generated by `yacc` and especially `lex` can have hundreds of unreachable `break` statements. The `-O` option in the C compiler often eliminates the resulting object code inefficiency. These unreachable statements are of little importance. There is usually nothing you can do about them, and the resulting messages would clutter up the `lint` output. If you want to get these messages, you can invoke `lint` without the `-b` option.

Function values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function "values" that are never returned. The `lint` program addresses these problems in a number of ways.

Locally, within a function definition, the appearance of both

```
return(expr);
```

and

```
return;
```

is cause for alarm. The `lint` program gives you the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by the control flow of a program reaching the end of the function. For example,

```
f (a) {
            if (a) return (3);
            g ();
}
```

In this example, if the result of `a` is false, `f` calls `g` and returns with no defined return value. This triggers a message from `lint`. If `g`, like `exit`, never returns, the message still is produced when, in fact, nothing is wrong. In practice, some potentially serious bugs have been discovered by using this feature.

On a global scale, `lint` detects cases where a function returns a value that is seldom or never used. When the value is never used, it can constitute an inefficiency in the function definition. When the value is seldom used, it can represent bad style (for example, not testing for error conditions).

The serious problem of using a function value when the function does not return one is also detected.

Type checking

The `lint` program enforces the C language type-checking rules more strictly than do the compilers. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure selection operators
- between the definition and uses of functions
- in the use of enumerations

There are several operators that have an implied balance between operand types. The assignment, conditional (`?:`), and relational operators have this property. The argument of a `return` statement and expressions used in initialization suffer similar conversions. In these operations, `char`, `short`, `int`, `long`, `unsigned`, `float`, and `double` types can be freely mixed.

The types of pointers must agree exactly except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type-checking rules also require that, in structure references, the left operand of the -> must be a pointer to structure; the left operand of the dot (.) must be a structure; and the right operand of both operators must be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types `float` and `double` can be freely matched, as can the types `char`, `short`, `int`, and `unsigned`. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are `=`, initialization, `==`, `!=`, function arguments, and return values.

If you want to turn off strict type checking for an expression, you should add the comment

```
/* NOSTRICT */
```

to the program immediately before the expression. This comment prevents strict type checking for the next line in the program only.

Type casts

The type-cast feature in the C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1;
```

where `p` is a character pointer. The `lint` program prints a message as a result of detecting this. Consider the assignment

```
p = (char *)1;
```

in which a cast is used to convert the integer to a character pointer. The programmer's intentions are clearly signaled. It seems harsh for `lint` to continue to print messages about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The `-c` flag controls the printing of comments about casts. When `-c` is in effect, casts are treated as though they were assignments subject to messages. Otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable character use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, `lint` prints messages about certain comparisons and assignments being illegal or nonportable. For example,

```
char c;
        ...
if ((c = getchar()) < 0)...
```

works on one machine but fails on machines whose characters always take on positive values. The real solution is to declare `c` an integer because `getchar` is actually returning integer values. In any case, `lint` prints the message

```
nonportable character comparison
```

A similar issue arises with bitfields. When constant values are assigned to bitfields, the field can be too small to hold the value. This is true especially because on some machines bitfields are considered signed quantities. While it might seem logical to consider that a two bitfield declared of type `int` cannot hold the value 3, the problem disappears if the bitfield is declared to have type `unsigned`.

Assignments of `long` to `int`

Bugs can arise from the assignment of `long` to `int`, which can truncate the contents. (Truncation happens only when `long` types hold a longer quantity than `int` types. In the current implementation, `long` types are the same length as `int` types.) This can happen in programs that are incompletely converted to use `typedef` statements. When a `typedef` variable is changed from `int` to `long`, the program can stop working. This is because some intermediate results can be assigned to `int` types, which are truncated. Because there are a number of legitimate reasons for assigning `long` types to `int` types, detecting these assignments is disabled by the `-a` option. If `lint` is using the `-p` option to detect possible portability problems, however, it can print the message

```
warning: conversion from long may lose accuracy
```

even if you're using the `-a` option.

Strange constructions

Several perfectly legal but somewhat strange constructions are detected by `lint`. The messages hopefully encourage better code quality and clearer style and can even point out bugs. The `-h` option is used to suppress the majority of these checks.

For example, in

```
*p++;
```

the `*` does nothing. This provokes the message

```
null effect
```

from `lint`. For another example,

```
unsigned x;
if(x < 0) ...
```

results in a test that never succeeds. For a third example,

```
unsigned x;
if(x > 0) ...
```

is equivalent to

```
if(x != 0)
```

which might not be the intended action. The `lint` program prints the message

```
degenerate unsigned comparison
```

in these latter two cases.

If a program contains something similar to

```
if( 1 != 0 ) ...
```

`lint` prints the message

```
constant in conditional context
```

because the comparison of 1 to 0 gives a constant result.

Another construction detected by `lint` involves operator precedence. Bugs that arise from misunderstandings about operator precedence can be compounded by spacing and formatting, making such bugs extremely hard to find. For example,

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to enclose such expressions in parentheses; `lint` encourages this with an appropriate message.

When the `-h` option is not used, `lint` prints messages about variables that are redeclared in inner blocks in a way that conflicts with their use in outer blocks. Although this is considered legal, it remains bad style, usually unnecessary, and frequently a bug.

Old syntax

Several forms of older syntax are now illegal. These fall into two classes: (1) assignment operators and (2) initialization.

The older forms of assignment operators (for example, `=+`, `=-`, and so on) can cause ambiguous expressions. For example,

```
a =-1;
```

can be taken as either

```
a =- 1;
```

or

```
a = -1;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (for example, `+=` and `-=`) have no such ambiguities. To encourage abandoning the older forms, `lint` prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1;
```

to initialize `x` to 1. This also caused syntactic difficulties. For example,

```
int x (-1);
```

looks somewhat like the beginning of a function definition

```
int x (y) { ...
```

The compiler must read past `x` to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equal sign between the variable and the initializer. For example,

```
int x = -1;
```

This is free of any possible syntactic ambiguity.

Pointer alignment

Certain pointer assignments can be reasonable on some machines and illegal on others, due entirely to alignment restrictions. The `lint` program tries to detect cases where such alignment problems might arise by finding pointers that are assigned to other pointers. This message appears:

```
possible pointer alignment problem
```

Multiple uses and side effects

In complicated expressions, the best order in which to evaluate subexpressions can depend on the machine being used. For example, on machines (like the PDP-11) in which the stack runs backward, function arguments are probably best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions might or might not be treated in a similar manner to ordinary arguments. The same uncertainty arises with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

To avoid compromising the efficiency of the C language on a particular machine, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers differ considerably in the order in which they evaluate complicated expressions. In particular, if any variable changed by a side effect is also used elsewhere in the same expression, the result is explicitly undefined.

The `lint` program checks for the important special case where a simple scalar variable is affected. For example,

```
a[i] = b[i++];
```

causes `lint` to print the message

```
warning: i evaluation order undefined
```

to call attention to this condition.

9 `sdb` Reference

This symbolic debugger is useful for debugging A/UX applications that do not call the A/UX Toolbox. Symbolic access to all variables is available, and procedures can be called directly from `sdb`. The `sdb` program is useful both for examining core images of cancelled programs and for providing an environment in which you can monitor and control the execution of a program. This debugger works on source code compiled with the A/UX compilers `c89`, `cc`, and `f77`.

The `sdb` program allows you to interact with a debugged program at the source language level. When debugging a core image from a cancelled program, `sdb` reports which line in the source program caused the error and allows symbolic access to all variables, displayed in the proper format.

You can place breakpoints at selected statements or step through the program line by line. To facilitate specification of lines in the program without a source listing, `sdb` provides a mechanism for examining the source text. You can call procedures directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines that provide formatted printouts of structured data.

Using sdb

To use sdb to its full capabilities, you need to compile the source program with the -g option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the -g option is specified, you can use sdb to trace the called functions at the time of the termination and to display the values of variables interactively.

A typical sequence of shell commands for debugging a core image is

```
cc -g prgm.c -o prgm
prgm
              Bus error - core dumped
sdb prgm
              main:25:    x[i] = 0;
*
```

In this example, the program prgm was compiled with the -g option and then executed. An error caused a core dump. The sdb program was then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function main at line 25 (line numbers are always relative to the beginning of the file) and displays the source text of the offending line. sdb then prompts you with an *, indicating that it awaits a command.

It is useful to know that sdb keeps track of the current function and current line. In this example, they are initially set to main and 25, respectively.

Arguments

In the above example, sdb was called with one argument, prgm. In general, sdb takes the following three arguments on the command line:

1. The name of the executable file to be debugged, which defaults to a.out when not specified. Even with the new COFF format, the executable file is named a.out. However, sdb does not work on old a.out format files. Only COFF files can be used with sdb.
2. The name of the core file, defaulting to core.
3. The name of the directory containing the source of the program being debugged.

The sdb program currently requires all source code to reside in a single directory. The default is the working directory. In the previous example, the default value for the second and third arguments was desired, so only the first argument was specified.

If an error occurs in a function that was not compiled with the -g option, sdb prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in main. The sdb program prints error message if main was not compiled with the -g option, but debugging can continue for those routines compiled with the -g option.

Example

The following example is a typical example of how sdb can be used. The first part of the example is the source file used to create the output file. The second part illustrates a sample session with sdb.

```
cat testdiv2.c
            main(argc, argv, envp)
            int argc;
            char **argv, **envp; {
                        int i;
                        i = div2(-1) ;
                        printf("-1/2 = %d\n", i);
            }
            div2(i)
            int i; {
                        int j;
                        j = i>>1;
                        return(j) ;
            }
cc -g testdiv2.c
a.out
            -1/2 = -1
```

With the output file created, sdb can now be invoked to find the problem:

Session / *Annotations*

```
sdb
 No core image                  warning message from sdb
 */^div2                        search for function div2
 7: div2(i) {                   it starts on line 7
*z                              print the next few lines
 7:          div2(i) {
 8:          int j;
 9:          j = i>>1;
 10:         return(j);
 11:         }
*div2:b                         place breakpoint at start of div2
 div2:9 b                       sdb echoes procedure name and line number
*r                              run the program
 a.out                          sdb echoes command line executed
 Breakpoint at                  execution stops just before line 9
 div:2:9:   j = i>>1;
*t                              print trace of subroutine calls
 div2(i=-1) [testdiv2.c:9]
 main (argc=1,...
*i/                             print i
 -1
*s                              single step
 div2:10:   return(j);          execution stops before line 10
*j/                             print j
 -1
*9d                             delete the breakpoint
```

(continued)➡

Session	*Annotations*
`*div2(1)/`	run `div2` with other arguments
` 0`	
`*div2(-2)/`	
` -1`	
`*div2(-3)/`	
` -2`	
`*q`	

Printing a stack trace

It's often useful to obtain a listing of the function calls that led to the error. You can do so with the `t` command. For example,

```
*t
 sub(x=2,y=3)              [prgm.c:25]
 inter(i=16012)            [prgrm.c:96
 main(argc=1,argv=0x7ffff54,
            envp=0x7ffff5c) [prgm.c:15]
```

This indicates that the error occurred within the function `sub` at line `25` in file `prgm.c`. The `sub` function was called with the arguments `x=2` and `y=3` from `inter` at line `96`. The `inter` function was called from `main` at line `15`. The main function is always called by the shell with three arguments often referred to as `argc`, `argv`, and `envp`. Note that `argv` and `envp` are pointers, so their values are printed in hexadecimal.

Examining variables

You can use the `sdb` program to display variables in the stopped program. To do so, type each name followed by a slash. For example,

```
*errflag/
```

causes `sdb` to display the value of variable `errflag`. Unless otherwise specified, variables are assumed to be local to or accessible from the current function. To specify a different function, use the form

```
*sub:i/
```

to display variable `i` in function `sub.f77` users can specify a common block variable in the same manner.

The `sdb` program supports a limited form of pattern matching for variable and function names. The symbol `*` is used to match any sequence of characters of a variable name; `?` matches any single character.

The following commands are useful examples of wildcards:

```
*x*/

*sub:y?

**/
```

The command `*x*/` prints the values of all variables beginning with `x`, the command `*sub:y?/` prints the values of all two-letter variables in function `sub` beginning with `y`, and the command `**/` prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command

```
**:*/
```

displays the variables for each function on the call stack.

The `sdb` program normally displays the variable in a format determined by its type, as declared in the source program. If you want to request a different format, place a specifier after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are as follows:

`b`	one byte
`h`	two bytes (half word)
`l`	four bytes (long word)

The lengths are effective with the formats `d`, `o`, `x`, and `u` only. If you don't specify a length, the word length of the host machine is used. A numeric length specifier can be used for the `s` or `a` commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies how many characters should be printed.

You can use a number of format specifiers with `sdb`, as follows:

`a`	print characters, starting at the variable address, until a null is reached
`c`	character
`d`	decimal
`f`	32-bit single-precision floating point
`g`	64-bit double-precision floating point
`i`	interpret as a machine-language instruction
`o`	octal
`p`	pointer to function
`s`	assume variable is a string pointer and print characters starting at the address pointed to by variable until a null is reached
`u`	decimal unsigned
`x`	hexadecimal

For example, the variable `i` can be displayed with

```
*i/x
```

which prints out the value of `i` in hexadecimal.

The `sdb` program also knows about structures, arrays, and pointers so that all of the following commands work:

```
*array[2][3]/

*sym.id/

*psym->usage/

*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Depending on your machine, you might be able to gain access only to one-dimensional arrays. Note that as a special case,

```
*psym->/d
```

displays the location pointed to by `psym` in decimal.

You also can display core locations by specifying their absolute addresses. The command

```
*1024/
```

displays location `1024` in decimal. As in the C language, numbers also can be specified in octal or hexadecimal, so the above command is equivalent to both

```
*02000/
```

and

```
*0x400/
```

It is possible to mix numbers and variables. For example,

```
*1000.x/
```

refers to an element of a structure starting at address `1000`, and

```
*1000->x/
```

refers to an element of a structure whose address is at `1000`. For commands of the type `*1000.x/` and `*1000->x/`, the `sdb` program uses the structure template of the last structure referenced.

The address of a variable is printed with the `=`. For example,

```
*i=
```

displays the address of `i`. Another useful feature, discussed more later, is the command

```
*./
```

which redisplays the last variable typed.

Display and manipulation

The `sdb` program is designed to make it easy for you to debug a program without constantly referring to a current source listing. Facilities are provided that perform context searches within the source files of the program you're debugging and display selected portions of the source files. The commands are similar to those of the A/UX system text editor `ed`(1). Like the editor, `sdb` keeps track of the current file and the current line within the file.

The `sdb` program also knows how the lines of a file are partitioned into functions, so it can find the current function. As noted elsewhere, the current function is used by a number of `sdb` commands.

Displaying the source file

There are four commands for displaying lines in the source file. They are useful for examining the source program and for determining the context of the current line. The commands are as follows:

`p`	Prints the current line.
`w`	Prints a window of ten lines around the current line.
`z`	Prints ten lines starting at the current line; this command also advances the current line by ten.
CONTROL-D	Scrolls and prints the next ten lines and advances the current line by ten; this command is used to display long segments of the program cleanly.

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file but also is used as input by some `sdb` commands.

Displaying another source file or function

The `e` command is used to display a different source file. Without any arguments, the `e` command prints the current function and filename. Additionally, these forms can be used:

```
*e function
*e file.c
```

The first form makes the file containing the named function the current file. The current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line becomes the first line of the named file.

Changing the current line display

The `z` and CONTROL-D commands have a side effect of making a new line the current line in the source file. The following paragraphs describe other commands that change the display.

There are two commands for searching for instances of regular expressions in source files, which are the following:

`*/regular expression/`	This command searches forward through the file for a line containing a string that matches the regular expression.
`*?regular expression?`	This command searches backward through the file for the same thing.

The trailing slash character (`/`) and question mark (`?`) can be omitted from these commands. Regular expression matching is identical to that of `ed`(1).

The `+` and `-` commands can be used to move the current line forward or backward by a specified number of lines. Typing a new line advances the current line by one and typing a number causes that line to become the current line in the file. These commands can be combined with the display commands. For example,

```
*+15z
```

advances the current line by 15 and then prints 10 lines.

A controlled testing environment

One very useful feature of `sdb` is **breakpoint** debugging. After entering `sdb`, certain lines in the source program can be specified to be breakpoints. The program is then started with the `sdb` command. The program is executed as normal until it's about to execute one of the breakpoints. The program stops and `sdb` reports the breakpoint where the program stopped. At this point, `sdb` commands can be used to display the trace of function calls and the values of variables. If you're satisfied the program is working correctly up to the breakpoint, you can delete some breakpoints and set others; then program execution can continue from the point at which it stopped.

A useful alternative to setting breakpoints is single stepping. You can request the `sdb` program to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified statement by statement.

If an attempt is made to single step through a function that is not compiled with the `-g` option, execution proceeds until a statement in a function compiled with the `-g` option is reached.

You also can have the program execute one machine-level instruction at a time. This is particularly useful when the program is not compiled with the `-g` option.

Setting and deleting breakpoints

You can set breakpoints at any line in a function that contains executable code. The command formats are as follows:

`*12b`	Sets a breakpoint at line `12` in the current file; line numbering starts at the beginning of the file, as printed by the source file display commands.
`*proc:12b`	Sets a breakpoint at line `12` of function `proc`.
`*proc:b`	Sets a breakpoint at the first line of `proc`.
`*b`	Sets a breakpoint at the current line.

You can delete breakpoints with the following commands:

`*12d`	Deletes a breakpoint at line `12` in the current file.
`*proc:12d`	Deletes a breakpoint at line `12` of function `proc`.
`*proc:d`	Deletes a breakpoint at the first line of `proc`.

In addition, you can use the command `d` to interactively delete the breakpoints. Each breakpoint location is printed, and a line is read from the user. You can delete a breakpoint if you specify a `y` or `d` at the beginning of the line.

The `B` command prints a list of the current breakpoints, and the `D` command deletes all breakpoints. `sdb` can automatically perform a sequence of commands at a breakpoint and then continue execution. You can do this with another form of the `b` command:

```
*12b t;x/
```

This causes both a trace back and the printing of value `x` each time execution gets to line `12`. The `a` command is a variation of the above command with the following two forms:

`*proc:12a`	prints the source line each time it is about to be executed
`*proc:a`	prints the function name and its arguments each time it is called

For both forms of the `a` command, execution continues after the function name or source line is printed.

Running the program

The `r` command begins program execution. It restarts the program as if it were invoked from the shell. The command

```
*r args
```

runs the program with the given arguments as if it were typed on the shell command line. If no arguments are specified, `sdb` uses the arguments from the last execution of the program. To run a program with no arguments, use the `R` command.

After the program starts, execution continues until a breakpoint is encountered, a signal such as interrupt or quit occurs, or the program terminates. In all cases, after an appropriate message is printed, control returns to `sdb`.

You can use the `c` command to continue executing a stopped program. A line number can be specified, as in

```
*proc:12c
```

This places a temporary breakpoint at the named line. `sdb` deletes the breakpoint when the `c` command finishes. There is also a `C` command that continues but passes back to the program the signal that stopped the program. This is useful for testing user-written signal handlers. Execution can be continued at a specified line with the `g` command. For example,

```
*17 g
```

continues at line `17` of the current function. Use this command to avoid executing a section of code that you know to be bad. You should not attempt to continue execution in a function other than the one in which the breakpoint is located.

Use the `s` command to run the program for a single line. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the `S` command. This command is like the `s` command, but does not stop within called functions. Use it when you're confident that the called function works correctly but you want to test the calling routine.

The `i` command runs the program one machine-level instruction at a time, while ignoring the signal that stopped the program. Its uses are similar to those of the `s` command. There is also an `I` command, which causes the program to execute one machine-level instruction at a time, but passes the signal that stopped the program back to the program.

Calling functions

You can call any of the program functions from `sdb`. This is useful both for testing individual functions with different arguments and for calling a function that prints structured data in a nice way. There are two ways to call a function:

```
*proc(arg1, arg2, ...)

*proc(arg1, arg2, ...)/m
```

The first simply executes the function. You can use the second format for calling functions; it executes the function and prints the value that it returns. The call prints the value in decimal format unless some other format is specified by `m`. Arguments to functions can be integer, character, or string constants or values of variables that are accessible from the current function.

If a function is called when the program isn't stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

Machine-language debugging

The `sdb` program has facilities for examining programs at the machine-language level. You can print the machine-language statements associated with a line in the source code, and you can place breakpoints at arbitrary addresses. You also can use the `sdb` program to display or modify the contents of the machine registers.

Displaying machine-language statements

To display the machine-language statements associated with line `25` in function `main`, use the command

```
*main:25?
```

The `?` command is identical to the `/` command except that it displays from text space. The default format for printing text space is the `i` format, which interprets the machine-language instructions. You can press CONTROL-D to print the next ten instructions.

You can specify absolute addresses instead of line numbers by appending a colon(`:`) to them. For example,

```
*0x1024:?
```

displays the contents of address `0x1024` in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line `0x1024` in the current function. You also can set or delete a breakpoint by specifying its absolute address. For example,

```
*0x1024:b
```

sets a breakpoint at address `0x1024`.

Manipulating registers

The `x` command prints the values of all the registers. Also, you can name individual registers instead of variables by appending a `%` to their names. For example,

```
*r3%
```

displays the value of register `r3`.

Other commands

Use the `q` command to exit `sdb`. The exclamation mark (`!`) command in `sdb` is identical to the same command in `ed`(1). It takes you to the shell, where you can execute a command.

You can change the values of variables when the program is stopped at a breakpoint. You can do this with the command

`*`*variable*`!` *value*

which sets the variable to the value you enter. The value can be a number, character constant, register, or the name of another variable. If the variable is of type `float` or `double`, it also can be a floating-point constant.

10 dbx Reference

This chapter describes the dbx debugger, a tool for source-level debugging and execution of C programs under A/UX.

The debugger operates by running the program being debugged as a child process. The debugger maintains control of the program being debugged by means of system hooks available through the ptrace(2) system call.

Using `dbx`

The `dbx` debugger can be used to symbolically debug all A/UX applications. It is particularly useful if the application makes calls to the A/UX Toolbox. The debugger can examine several kinds of code, including Macintosh and COFF binaries.

For C code compiled with the appropriate options, `dbx` can provide the following functions:

- examination of the symbol table
- variable, expression, and condition tracing
- function and procedure tracing
- source-line tracing
- signal trapping
- variable assignment
- step-by-step execution
- variable-printing and expression-printing capabilities
- real-time editing capability

`dbx` also has the capability of examining object code at the machine-language level and its machine-level facilities can be used on any program. The ability to examine machine language is useful when you don't have the source code for a program or when you want to inspect compiled assembly code to see exactly what the compiler and optimizer did to your source code.

`dbx` syntax

The command-line syntax for `dbx` is

`dbx [-r] [-i] [-D] [-I` *dir*`] [-c` *file*`] [`*objfile* `[`*coredump*`]]`

The *objfile* is an object file produced by the `c89`, `cc`, or `f77` compilers. To use symbolic debugging, the object file must be created using the `-g` compiler command option. Object files created with the `-g` option contain a symbol table that includes the names of all the source files translated by the compiler to create it. These files are available for examination while using the debugger. Files created without the `-g` option can be debugged, but the symbol-table information is not available. Object files created with the `-c` option are intermediate relocatable object code files, which can be examined but not run. (Such files are called "dot-oh" files (`.o`) after the extension appended to the filename.)

◆ **Note** Optimized code cannot be symbolically debugged with `dbx`; the code optimizer is disabled when the `-g` option of the C compiler is used. ◆

If a file named `core` exists in the current directory or a *coredump* file is specified, `dbx` can be used to examine the state of the program when it faulted.

If the file `.dbxinit` exists in the current directory, the debugger commands in it are executed. `dbx` also checks for a `.dbxinit` in the user's home directory, if there isn't one in the current directory.

The command line options and their meanings are as follows:

`-c` *file*	Execute the `dbx` commands in the file before reading from standard input.
`-i`	Force `dbx` to act as though standard input is a terminal.
`-r`	Execute *objfile* immediately. (If it terminates successfully, `dbx` exits. Otherwise, `dbx` reports the reason for termination and you are offered the option of entering the debugger or letting the program fault. `dbx` reads from `/dev/tty` when `-r` is specified and standard input is not a terminal.)
`-D`	Add Macintosh OS trap names; this is used in debugging Macintosh hybrid applications.
`-I` *dir*	Add *dir* to the list of directories that are searched when looking for a source file. (Normally `dbx` looks for source files in the current directory and in the directory where *objfile* is located. The directory search path also can be set within `dbx` with the `use` command.)

Unless `-r` is specified, `dbx` just prompts using the prompt `(dbx)` and waits for a command.

Example

The following example shows first a C source-code file and then the invocation of the `dbx` debugger.

```
/*this is a C source code file, hello.c */
#include <stdio.h>

int global1;
int global2 = 2;

main (char *argc[], int argv, char *envp[])
{
            register int a;
            int b;

            a = 10;
            b = 20 + global2;
            printme (a, b);
            exit (0);
}

static printme (int a, int b)
{
            printf ("hello world %d %d \n", a, b);
}
```

To use the debugger on this file, you first compile the file,

```
cc -g hello.c -o hello
```

then invoke the debugger

```
dbx hello
```

The debugger responds with something like

```
dbx version 3.6 of 5/15/91 21:01 (salmon).

Type 'help' for help.

reading symbolic information ...
```

The debugger is now ready to accept commands.

Command list

Table 10-1 shows an alphabetical list of `dbx` commands. The following sections group the commands by function and describe them in detail.

Table 10-1 `dbx` commands

`alias`	`func`	`run`	`tracei`
`assign`	`help`	`set`	`unalias`
`call`	`ignore`	`sh`	`unset`
`catch`	`list`	`source`	`up`
`cont`	`next`	`status`	`use`
`delete`	`nexti`	`step`	`whatis`
`down`	`print`	`stepi`	`where`
`dump`	`quit`	`stop`	`whereis`
`edit`	`rerun`	`stopi`	`which`
`file`	`return`	`trace`	

Execution and tracing commands

You can use a variety of commands (discussed in the following list) to see how the program executes. breakpoints can be set in several ways: `dbx` can stop at a certain source-line number, at a certain signal, when a procedure or function is called, when a variable is changed, or when a condition becomes true.

`run` [*args*] [<*filename*] [>*filename*]
`rerun` [*args*] [<*filename*] [>*filename*]
These commands start executing *objfile*, passing *args* as command-line arguments; the characters `<` and `>` can be used to redirect output and input in the usual manner. When `rerun` is used without any arguments, the previous argument list is passed to the program; otherwise it is identical to `run`.

`trace [in` *procedure/function*`] [if` *condition*`]`
`trace` *source-line-number* `[if` *condition*`]`
`trace` *procedure/function* `[in` *procedure/function*`] [if` *condition*`]`
`trace` *expression* `at` *source-line-number* `[if` *condition*`]`
`trace` *variable* `[in` *procedure/function*`] [if` *condition*`]`

These commands have tracing information printed when the program is executed. A number is associated with the command that can be used with the `delete` command to turn off the tracing.

The first argument describes what is to be traced. If it is a source-line number, `dbx` prints the line immediately prior to being executed. Source-line numbers in a file other than the current one must be preceded by the name of the file in double quotation marks and a colon, for example,

```
"yoyodyne.c":21
```

If the argument is a procedure or function name, every time it is called `dbx` prints information telling what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted. If the argument is a function, `dbx` also prints the value that *function* returns.

If the argument is an expression with an `at` clause, `dbx` prints the value of the expression whenever the identified source line is reached.

If the argument is a variable, `dbx` prints the name and value of the variable whenever it changes. Execution is substantially slower during this form of tracing.

If no argument is specified, `dbx` prints all source lines before executing them. Execution is substantially slower during this form of tracing.

The clause "`in` *procedure/function*" restricts tracing information to be printed only while executing inside the given procedure or function.

The term *condition* is a Boolean expression and is evaluated prior to printing the tracing information; if it is false, the information is not printed.

`stop if` *condition*
`stop at` *source-line-number*`[if` *condition*`]`
`stop in` *procedure/function*`[if` *condition*`]`
`stop` *variable*`[if` *condition*`]`

These commands stop execution when the given line is reached, procedure or function is called, variable is changed, or condition becomes true. Execution can be resumed with the `cont` command.

`status` [> *filename*]
This command prints the currently active `trace` and `stop` commands.

`delete` *command-number* ...
This command removes traces or stops corresponding to the given numbers. The `status` command prints the numbers associated with traces and stops.

`catch` *number*
`catch` *signal-name*
`ignore` *number*
`ignore` *signal-name*
These commands start or stop trapping a signal before it is sent to the program. This is useful when a program being debugged handles signals such as interrupts. A signal can be specified by number or by a name (for example, `SIGINT`). Signal names are not case sensitive and the `SIG` prefix is optional. By default, all signals are trapped except `SIGCONT`, `SIGCHILD`, `SIGALRM`, and `SIGKILL`.

`cont` *integer*
`cont` *signal-name*
These commands continue execution from where the process stopped. If a signal is specified, the process continues as though it received the signal. Otherwise, the process is continued as though it had not stopped.

Execution cannot be continued if the process finishes, that is, if it called `exit`. Even if this call is made, however, the user can examine the program state because `dbx` does not allow the process to actually exit.

`step`
This command executes one source line.

`next`
This command executes up to the next source line. The difference between `next` and `step` is that if the line contains a call to a procedure or function, the `step` command stops at the beginning of that block, while the `next` command does not.

`return [`*procedure*`]`
This command continues until the named procedure is returned to, or until the current procedure returns, if none is specified.

`call` *procedure*`(`*parameters*`)`
This command executes the object code associated with the named procedure or function.

Example

The following examples of `dbx` sessions demonstrate the various commands. To begin the examples for this chapter, the source is listed first so you know the number of the program assigned to each source line. (The `list` command is covered in the section "Accessing Source Files," later in this chapter.)

```
(dbx) list 1,100
     1   #include <stdio.h>
     2
     3   int global1;
     4   int global2 = 2;
     5
     6   main (char *argc[], int argv, char *envp[])
     7
     8   {
     9       register int a;
    10       int b;
    11
    12       a = 10;
    13       b = 20 + global2;
    14       printme (a, b);
    15       exit (0);
    16   }
    17
    18   static printme (int a, int b)
```

```
   19
   20   {
   21       printf ("hello world %d %d \n", a, b);
   22   }
```

The `run` command executes the program.

```
(dbx) run

hello world 10 22
```

The `stop` command sets breakpoints at source lines `12` and `14`, and the `status` command checks which commands are active.

```
(dbx) stop at 12
[1] stop at "hello.c":12
(dbx) stop at 14
[2] stop at "hello.c":14
(dbx) status
[1] stop at "hello.c":12
[2] stop at "hello.c":14
```

The `delete` command removes the breakpoint at source line `12`, and again the `status` command checks which commands are active.

```
(dbx) delete 1
(dbx) status
[2] stop at "hello.c":14
```

The `run` command executes the program until the breakpoint is reached. The program stops and reports the source line at which it stopped. The `cont` command is subsequently used to resume execution.

```
(dbx) run
[2] stopped in main at line 14 in file "hello.c"
   14         printme (a, b);
(dbx) cont
hello world 10 22

execution completed
```

You also can use the `step` command to single step through the execution after a breakpoint.

```
(dbx) run
[2] stopped in main at line 14 in file "hello.c"
   14        printme (a, b);
(dbx) step
stopped in printme at line 20 in file "hello.c"
   20   {
(dbx) step
stopped in printme at line 21 in file "hello.c"
   21        printf ("hello world %d %d \n", a, b);
(dbx) step
hello world 10 22

program exited
```

The `trace` command prints information as the program is executing. Again, the status command reports all `stop` and `trace` settings.

```
(dbx) trace
[5] trace
(dbx) status
[2] stop at "hello.c":14
[5] trace
```

When a `trace` command is given without arguments, each source line is printed immediately before it is executed.

```
(dbx) run
entering function main
trace:      8   {
trace:     12          a = 10;
trace:     13          b = 20 + global2;
trace:     14          printme (a, b);
[2] stopped in main at line 14 in file "hello.c"
   14                  printme (a, b);
(dbx) cont
```

```
entering function printme
trace:     20    {
trace:     21          printf ("hello world %d %d \n", a, b);
hello world 10 22

program exited
```

You also can `trace` functions, variables, expressions, and specific source lines.

```
(dbx) status
[5] trace
(dbx) delete 5
(dbx) trace printme
[8] trace printme
(dbx) run
calling printme(a = 10, b = 22) from function main
hello world 10 22
returning 19 from printme

execution completed
```

Printing variables and expressions

Names are resolved, first using the static scope of the current function and then using the dynamic scope if the name is not defined in the static scope. If static and dynamic searches do not yield a result, `dbx` chooses an arbitrary symbol and prints the message

`[using` *qualified name*`]`

The name resolution procedure can be overridden by qualifying an identifier with a block name, for example, *module.variable*. For C, `dbx` treats source files as modules named by the filename without the usual `.c` suffix.

Expressions are specified with an approximately common subset of C and Pascal syntax. Indirection can be denoted using either a prefix `*` or a suffix `^`; array subscripts are enclosed by brackets (`[ ]`). The field reference operator (`.`) can be used with pointers as well as records, making the C operator `->` unnecessary (although it is supported).

Types of expressions are checked; the type of an expression can be overridden by using *type-name*`(`*expression*`)`. When there is no corresponding named type, the special constructs `&`*type-name* and `$$`*tag-name* can be used to represent a pointer to a named type or C structure tag.

`assign` *variable* `=` *expression*
Assign the value of the expression to the variable.

`dump [`*procedure*`] [>`*filename*`]`
Print the names and values of variables in the given procedure, or the current one if none is specified. If the procedure given is "`.`", all the active variables are dumped.

`print` *expression* `[,` *expression* ...`]`
Print the values of the expressions.

`whatis` *name*
Print the declaration of the given name, which can be qualified with block names as explained earlier in this section.

`which` *identifier*
Print the full qualification of the given identifier, that is, the outer blocks with which the identifier is associated.

`up [`*count*`]`
`down [`*count*`]`
Move the current function, which is used for resolving names, up or down the stack count levels. The default count is `1`.

`where`
Print a list of the active procedures and functions and the argument passed to them.

`whereis` *identifier*
Print the full qualification of all the symbols whose name matches the given identifier. The order in which the symbols are printed is not meaningful.

Example

To show the effect of some of these commands, the following example sets a breakpoint and runs the program:

```
(dbx) stop at 13
[1] stop at "hello.c":13
(dbx) run
[1] stopped in main at line 13 in file "hello.c"
   13        b = 20 + global2;
```

The `assign` command lets you change the value of a variable; the `print` command displays the value of variables of expressions. Note that at the breakpoint, `b` is not assigned a value because the program stops before the source line is executed.

```
(dbx) assign a = 80
(dbx) print a, b
80 0
(dbx) cont
hello world 80 22

execution completed
```

The `where` command prints a list of the active functions and the arguments passed to them.

```
(dbx) run
[1] stopped in main at line 13 in file "hello.c"
   13        b = 20 + global2;
(dbx) where
main(argc = 0x1, argv = 4294966980, envp = 0xfffffecc), line
13 in "hello.c"
_start() at 0x10b
```

The `whereis` command prints a list of all the functions its argument is located within.

```
(dbx) whereis a
hello.printme.a hello.main.a
```

The `dump` command prints the names and values of all variables in a given function (or the current function if none is named).

```
(dbx) dump
main(argc = 0x1, argv = 4294966980, envp = 0xfffffecc), line
13 in "hello.c"
b = 0
a = 10
(dbx) assign a = 80
(dbx) dump
main(argc = 0x1, argv = 4294966980, envp = 0xfffffecc), line
13 in "hello.c"
b = 0
a = 80
```

The `whatis` command prints the declaration of the variable or function you provide as an argument to the command.

```
(dbx) whatis a
register int a;
(dbx) whatis printme
int printme(a, b)
int a;
int b;
```

Accessing source files

`/`*regular expression*`[/]`
`?`*regular expression*`[?]`
Search forward or backward in the current source file for the given pattern.

`edit` [*filename*]
`edit` *procedure/function-name*
Invoke an editor on *filename* or the current source file if no *filename* is specified. If a procedure or function name is specified, the editor is invoked on the file that contains it. Which editor is invoked by default depends on the installation. You can override the default by setting the environment variable `EDITOR` to the name of the desired editor.

`file` [*filename*]
Change the current source filename to *filename*. If none is specified, the current source filename is printed.

`func` [*procedure/function*]
Change the current function. If none is specified, print the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

`list` [*source-line-number* [, *source-line-number*]]
`list` *procedure/function*
List the lines in the current source file from the first line number to the second, inclusive. If no lines are specified, the next `$listwindow` lines are listed (the default is 10). If the name of a procedure or function is given, lines *n-k* to *n+k* are listed, where *n* is the first statement in the procedure or function and *k* is defined by the value `$listwindow`. An example of the list command is shown in the example portion of the section "Execution and Tracing Commands," earlier in this chapter.

`use` *directory-list*
Set the list of directories to be searched when looking for source files.

Command aliases and variables

When commands are processed, `dbx` first checks to see whether the word is an alias for either a command or a string. If it is an alias, then `dbx` treats the input as though the corresponding string (with values substituted for any parameters) were entered. The formats for aliases are as follows:

`alias` *name name*

`alias` *name string*

`alias` *name (parameters) string*

For example, to define an alias `rr` for the command `rerun`, you can use the command

```
alias rr rerun
```

To define an alias called `b` that sets a stop at a particular line, you can use the command

```
alias b(x) "stop at x"
```

The command `b(12)` subsequently expands to `stop at 12`.

`set` *name* `[=` *expression*`]`
The `set` command defines values for debugger variables. The names of these variables cannot conflict with names in the program being debugged and are expanded to the corresponding expression within other commands. The following variables have a special meaning:

`$hexchars` `$hexints` `$hexoffsets` `$hexstrings`	When these variables are set, `dbx` prints out characters, integers, offsets from registers, or character pointers, respectively, in hexadecimal.
`$listwindow`	The value of this variable specifies the number of lines to list around a function or when the `list` command is given without any parameters. Its default value is 10.

`$unsafecall` `$unsafeassign`	When `$unsafecall` is set, strict type checking is turned off for arguments to subroutine calls or function calls (for example, in the `call` statement). When `$unsafeassign` is set, strict type checking between the two sides of an assign statement is turned off. These variables should be used only with great care, because they severely limit the usefulness of `dbx` for detecting errors.
`unalias` *name*	Remove the alias for *name*.
`unset` *name*	Delete the debugger variable associated with *name*.

Machine-level commands

`tracei [`*address*`] [if` *cond*`]`
`tracei [`*variable*`] [at` *address*`] [if` *cond*`]`
`stopi [`*address*`] [if` *cond*`]`
`stopi [at] [`*address*`] [if` *cond*`]`
Turn on machine-level tracing or set a stop using a machine-instruction address.

`stepi`
`nexti`
Execute a single step as in `step` or `next`, but do a single instruction rather than a source line.

address , *address*/ [*mode*]
address / [*count*] [*mode*]
Print the contents of memory starting at the first address and continuing up to the second address or until *count* items are printed. If the address is " . ", the address following the one printed most recently is used. The mode specifies how memory is to be printed; if it is omitted, the previous mode specified is used. The initial mode, `X`, prints a long word in hexadecimal. The following modes are supported:

`i`	print the machine instruction
`d`	print a short word in decimal

`D`	print a long word in decimal
`o`	print a short word in octal
`O`	print a long word in octal
`x`	print a short word in hexadecimal
`X`	print a long word in hexadecimal
`b`	print a byte in octal
`c`	print a byte as a character
`s`	print a string of characters terminated by a null byte
`f`	print a single-precision real number
`g`	print a double-precision real number

Symbolic addresses are specified by preceding the name with an `&`. MC68000-family registers are denoted by `$d`*n* for the data registers, `$a`*n* for the address registers, and `$fp`*n* for the floating-point registers, where *n* is the number of the register. Addresses can be expressions made up of other addresses and the operators `+`, `-`, and indirection (unary `*`).

`help`
Print out a synopsis of `dbx` commands.

`quit`
Exit `dbx`.

`sh` *command-line*
Pass the command line to the shell for execution. The `SHELL` environment variable determines which shell is used.

`source` *filename*
Read `dbx` commands from the given filename.

Example

The following is an example of the `tracei` command in which the breakpoints are cleared (though this is not necessary), the `tracei` command given, and the program run. The debugger prints all the assembly-language statements just before they are executed.

```
(dbx) status
[1] stop at "hello.c":13
(dbx) delete 1
(dbx) tracei
[4] tracei
(dbx) run
tracei: 000000d6        subq.l      &0x8,%sp
tracei: 000000d8        mov.l       0x8(%sp),(%sp)
tracei: 000000dc        lea         0xc(%sp),%a0
tracei: 000000e0        mov.l       %a0,0x4(%sp)
tracei: 000000e4        mov.l       %a0,%a1
tracei: 000000e6        tst.l       (%a0)+
tracei: 000000e8        bne.b       0xe6
tracei: 000000e6        tst.l       (%a0)+
tracei: 000000e8        bne.b       0xe6
tracei: 000000ea        mov.l       %a0,0x8(%sp)
tracei: 000000ee        mov.l       %a0,0x400988
tracei: 000000f4        jsr         initfpu
tracei: 000000fa        jsr         __istart
tracei: 00000100        jsr         __compatmode
tracei: 00000106        jsr         main
```

(continued)➡

```
entering function main
tracei: 00000118          link.l     %a6,&0xfffffff4
tracei: 0000011e          movm.l     &<d2,d3>,0x4(%sp)
tracei: 00000124          fmovm.x    &<>,(0xc,%a7)
tracei: 0000012e          mov.l      &0xa,%d2
tracei: 00000130          mov.l      0x4009b0,%d0
tracei: 00000136          add.l      &0x14,%d0
tracei: 0000013c          mov.l      %d0,%d3
tracei: 0000013e          mov.l      %d3,-(%sp)
tracei: 00000140          mov.l      %d2,-(%sp)
tracei: 00000142          bsr  printme
entering function printme
tracei: 00000166          link.l       %a6,&0xfffffffc
tracei: 0000016c          movm.l       &<>,0x4(%sp)
tracei: 00000172          fmovm.x      &<>,(0x4,%a7)
tracei: 0000017c          mov.l        0xc(%fp),-(%sp)
tracei: 00000180          mov.l        0x8(%fp),-(%sp)
tracei: 00000184          mov.l        &0x4009b4,-(%sp)
tracei: 0000018a          bsr.l        printf
hello world 10 22
program exited
```

You terminate `dbx` with the `quit` command:

```
(dbx) quit
```

11 `f77` Command Syntax

This chapter describes how to invoke and use the A/UX Fortran 77 compiler.

The `f77` command compiles and loads Fortran and Fortran-related files into an executable module. The `f77` command invokes the C compiler to translate C source files and the assembler to translate assembler source files. If EFL (compiler) source files are given as arguments to the `f77` command, they are translated into Fortran before being presented to this Fortran compiler (see Chapter 13, "ELF Reference"). Object files are link-edited unless the `-c` option is used.

◆ **Note** The `f77` and `cc` commands have slightly different link-editing sequences. Fortran programs need two extra libraries, `libI77.a` and `libF77.a`, and an additional startup routine. ◆

Using `f77`

The command to run the A/UX Fortran compiler is

`f77 [`*option* ... `] [`*file*`]`

The following options have the same meaning in the Fortran compiler as in `cc`(1) (see `ld`(1) for load-time options):

`-A` *factor*	Expand the default symbol table allocations for the assembler and link editor. (The default allocation is multiplied by the factor given.)
`-c`	Suppress loading and produce `.o` files for each source file.
`-g`	Have the compiler produce additional symbol-table information for `sdb`(1); also pass the `-lg` flag to `ld`(1).
`-w`	Suppress all warning messages. (If the option is `-w66`, only Fortran 66 compatibility warnings are suppressed.)
`-p`	Prepare object files for profiling (see `prof`(1)).
`-O`	Invoke an object-code optimizer.
`-S`	Compile the named programs, and leave the assembler-language output on corresponding files with a `.s` suffix (no `.o` is created).
`-o` *output*	Name the final output file *output* instead of `a.out` (default).

The following options are specific to `f77`:

`-onetrip`	Compile `do` loops that are performed at least once if reached. (Fortran 77 `do` loops are not performed at all if the upper limit is smaller than the lower limit.)
`-u`	Make the default type of a variable `undefined` rather than using the default Fortran rules.
`-C`	Compile code to check that subscripts are within declared array bounds.
`-F`	Apply EFL preprocessor to relevant files. (Put the result in the file with the extension changed to `.f`, but do not compile.)
`-m`	Apply the M4 preprocessor to each `.e` file before transforming it with the EFL preprocessor.

`-N` *table entries*	Set the maximum number of table entries to the number *entries*. Replace *table* with one of the following letter designations corresponding to a compiler table:

`q`	equivalence table
`x`	external names table
`s`	statement number table
`c`	control block table
`n`	identifier table

To allow up to 1000 statement numbers, use the option with these arguments:

```
-Ns1000
```

`-E` *x*	Use the string *x* as an EFL option in processing `.e` files.

Other arguments are taken to be loader option arguments, `f77`-compatible object programs (typically produced by an earlier run), or libraries of `f77`-compatible routines. These programs, together with the results of any specified compilations, are loaded (in the order given) to produce an executable program with the name `a.out` (default).

The *file* argument to `f77` can have one of the following suffixes:

`.f`	Fortran source file
`.e`	EFL source file
`.c`	C language source file
`.s`	assembler source file
`.o`	object file

Arguments are processed as follows:

- Arguments whose names end with `.f` are taken to be Fortran 77 source programs (When compiled, a source program produces an object file with the same root name, but with a `.o` substituted for the `.f` extension.)
- Arguments whose names end with `.e` are taken to be EFL source programs
- Arguments whose names end with `.c` or `.s` are taken to be C or assembly source programs, respectively, and are compiled or assembled, producing a `.o` file

Related utilities

These utilities are useful adjuncts to `f77`. Their special characteristics are described in the following list:

`efl`	Compiles a program written in Extended Fortran Language (EFL) into Fortran 77. (See Chapter 13, "ELF Reference," for information on how to use this command.)
`asa`	Interprets the output of Fortran programs that use ASA carriage control characters. (See `asa`(1) for information on how to use this command.)
`fsplit`	Splits the named file(s) into separate files, with one procedure per file. (See `fsplit`(1) for information on how to use this command.)

12 Fortran Language Reference

This chapter describes the Fortran 77 run-time system and language as implemented on the A/UX system. Also described are the interfaces between procedures and the file formats assumed by the I/O system.

Please note that this chapter only describes the differences between the A/UX Fortran 77 and the ANSI Standard Fortran 77 and is not intended to be a complete language reference.

Fortran standards

Fortran 77 and Fortran 66 are names for two standardized versions of the language.

Fortran 77 includes almost all of Fortran 66. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O.

The `f77` language described in this chapter is an extended version of a Fortran 77 standard language, as specified in *ANSI Standard X3.9-1978 Fortran*.

Most of the extensions included in `f77` are useful additions; however, some are necessary to facilitate communication with C language functions, allowing easier compilation of old (Fortran 66) programs.

Language extensions

`double complex` data type

In the `double complex` data type, each datum is represented by a pair of double-precision real variables. A `double complex` version of every complex built-in function is provided.

Internal files

The Fortran 77 American National Standard introduces internal files (memory arrays) but restricts their use to `formatted sequential` I/O statements. The A/UX I/O system also permits internal files to be used in `direct` and `unformatted reads` and `writes`.

`implicit undefined` statement

Fortran has a rule that the variable type that does not appear in a type statement is `integer` if its first letter is `i`, `j`, `k`, `l`, `m`, or `n`. Otherwise, it is `real`. Fortran 77 has an `implicit` statement for overriding this rule. An additional type statement, `undefined`, is permitted. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism. The compiler issues a diagnostic for each variable that is used but does not appear in a type statement. Specifying the `-u` compiler option is equivalent to beginning each procedure with this statement.

Recursion

Procedures can call themselves directly or through a chain of other procedures. This differs from ANSI Standard Fortran 77, which does not allow any form of recursion.

Automatic storage

`static` and `automatic` are recognized keywords in this implementation, but not in ANSI Standard Fortran 77. These keywords can appear in `implicit` statements or as *types* in type statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared automatic for each invocation of the procedure. Automatic variables cannot appear in `equivalence`, `data`, or `save` statements.

Variable length input lines

The Fortran 77 American National Standard expects input to the compiler to be in a 72-column format (except in comment lines).

- The first five characters are the statement number.
- The next character is the continuation character.

- The next 66 are the body of the line.
- If there are fewer than 72 characters on a line, the compiler pads it with blanks.
- Characters after the first 72 are ignored.

To make it easier for you to type in Fortran programs, this compiler also accepts input in variable length lines:

- An ampersand (`&`) in the first position of a line indicates a continuation line; the remaining characters form the body of the line.
- A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line.
- A tab anywhere except in one of the first six positions on the line is treated as another kind of blank by the compiler.

Uppercase/lowercase

In the Fortran 77 Standard, there are only 26 letters because Fortran is a one-case language. This compiler expects lowercase input.

By default, the compiler converts all uppercase characters to lowercase except those inside character constants. If you specify the `-U` compiler option, uppercase letters are not transformed. In this mode, you can specify external names that have uppercase letters and you can have distinct variables differing in case only.

If the `-U` option is set, keywords are recognized only if they appear in lowercase.

`include` statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file `stuff`. The `include` statements can be nested to a reasonable depth, currently ten.

Binary initialization constants

A `logical`, `real`, or `integer` variable can be initialized in a data statement by a **binary constant,** which is denoted by a letter, followed by a quoted string. If the letter is `b`, the string is binary and only zeros and ones (0 and 1) are permitted. If the letter is `o`, the string is octal, with digits zero through seven (0–7). If the letter is `z` or `x`, the string is hexadecimal, with digits zero through nine (0–9), `a` through `f`. Thus, the statements

```
integer a(3)
data a/b'1010',o'12',z'a'/
```

initialize all three elements of `a` to `10`.

Character strings

To be compatible with the C language, this compiler recognizes the following backslash escapes:

`\n`	newline
`\t`	tab
`\b`	backspace
`\f`	form feed
`\0`	null
`\'`	apostrophe (does not terminate a string)
`\"`	quotation mark (does not terminate a string)
`\\`	\ (backslash)
`\x`	the character (in general)

Fortran 77 has only one quoting character: the apostrophe (`'`). This compiler and I/O system recognize both the apostrophe and the double quote (`"`). If a string begins with one variety of quotation mark, you can embed the other within it without using the repeated quote or backslash escapes.

Character string constants and scalar local character variables (except those declared with an `equivalence` statement) are aligned to `integer` word boundaries. Each character string constant appearing outside a `data` statement is followed by a null character to ease communication with C language routines.

Hollerith

Fortran 77 does not have the old Hollerith (*n*h) notation, although the new Standard recommends implementing it to improve compatibility with old programs. In this compiler, Hollerith data can be used in place of character string constants and also can be used to initialize noncharacter variables in data statements.

Equivalence statements

This compiler permits single subscripts in `equivalence` statements under the interpretation that all missing subscripts are equal to one. A warning message is printed for each such incomplete subscript.

One-trip `do` loops

The Fortran 77 American National Standard requires that the range of a `do` loop not be performed if the initial value is already past the limit value. For example,

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a `do` loop would be performed at least once.

To accommodate old programs, although they are in violation of the 1977 Standard, this compiler offers the `-onetrip` compiler option, which causes loops whose initial value is greater than or equal to the limit value to be performed exactly once.

Commas in formatted input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile to do so. When you request a formatted read of noncharacter variables, commas can be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, if you have the format

```
(i10, f20.10, i4)
```

the record

```
-345,.05e-3,12
```

is read correctly.

Short integers

This compiler accepts declarations of type `integer*2`. (Ordinary integers follow the Fortran rules about occupying the same space as a `real` variable; they are assumed to be of C language type `long int`; half word integers are of C language type `short int`.) An expression involving only objects of type `integer*2` is also of that type. Generic functions return short or long integers, depending on the actual types of their arguments. If a procedure is compiled using the `-I2` flag, all small integer constants are of type `integer*2`. If the precision of an integer-valued intrinsic function cannot be determined by the generic function rules, the compiler chooses one that returns the prevailing length (`integer*2` when the `-I2` command flag is in effect). When the `-I2` option is in effect, all quantities of type `logical` are deemed `short`. Note that these `short integer` and `logical` quantities do not obey the standard rules for storage association.

Additional intrinsic function library

This compiler supports all the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (`or`, `and`, `xor`, and `not`) and for accessing command arguments (`getarg` and `iargc`).

The following is the Fortran intrinsic function library plus some additional functions. These functions are automatically available to the Fortran programmer and require no special invocation of the compiler. The dagger (†) beside some of the commands indicates that they are not part of ANSI standard F77. In parentheses beside each function description is the location for the command in *A/UX Programmer's Reference.*

These functions are as follows:

†`abort`	terminate program (`abort`(3F))
`abs`	absolute value (`max`(3F))
`acos`	arccosine (`acos`(3F))
`aimag`	imaginary part of complex argument (`aimag`(3F))
`aint`	integer part (`aint`(3F))
`alog`	natural logarithm (`log`(3F))
`alog7`	common logarithm (`alog10`(3F))
`amax0`	maximum value (`max`(3F))
`amax1`	maximum value (`max`(3F))
`amin0`	minimum value (`min`(3F))
`amin1`	minimum value (`min`(3F))
`amod`	`mod`(3F)
†`and`	bitwise Boolean (`bool`(3F))
`anint`	nearest integer (`round`(3F))
`asin`	arcsine (`asin`(3F))
`atan`	arctangent (`atan`(3F))
`atan2`	arctangent (`atan2`(3F))
`cabs`	complex absolute value (`abs`(3F))
`ccos`	complex cosine (`cos`(3F))
`cexp`	complex exponential (`exp`(3F))
`char`	explicit type conversion (`ftype`(3F))
`clog`	complex natural logarithm (`log`(3F))
`cmplx`	explicit type conversion (`ftype`(3F))
`conjg`	complex conjugate (`conjg`(3F))
`cos`	cosine (`cos`(3F))

`cosh`	hyperbolic cosine (`cosh`(3F))
`csin`	complex sine (`sin`(3F))
`csqrt`	complex square root (`sqrt`(3F))
`dabs`	absolute value (`abs`(3F))
`dacos`	arccosine (`acos`(3F))
`dasin`	arcsine (`asin`(3F))
`datan`	arctangent (`atan`(3F))
`datan2`	double-precision arctangent (`atan2`(3F))
`dble`	explicit type conversion (`ftype`(3F))
†`dcmplx`	explicit type conversion (`ftype`(3F))
†`dconjg`	complex conjugate (`conjg`(3F))
`dcos`	cosine (`dcos`(3F))
`dcosh`	hyperbolic cosine (`cosh`(3F))
`ddim`	positive difference (`dim`(3F))
`dexp`	exponential (`exp`(3F))
`dim`	positive difference (`dim`(3F))
†`dimag`	imaginary part of complex argument (`aimag`(3F))
`dint`	integer part (`aint`(3F))
`dlog`	natural logarithm (`log`(3F))
`dlog10`	common logarithm (`log10`(3F))
`dmax1`	maximum value (`max`(3F))
`dmin1`	minimum value (`min`(3F))
`dmod`	remaindering (`dmod`(3F))
`dnint`	nearest integer (`round`(3F))
`dprod`	double-precision product (`dprod`(3F))
`dsign`	transfer of sign (`sign`(3F))
`dsin`	sine (`sin`(3F))
`dsinh`	hyperbolic sine (`sinh`(3F))
`dsqrt`	square root (`sqrt`(3F))

`dtan`	tangent (`tan`(3F))
`dtanh`	hyperbolic tangent (`tanh`(3F))
`exp`	exponential (`exp`(3F))
`float`	explicit type conversion (`ftype`(3F))
†`getarg`	return command-line argument (`getarg`(3F))
†`getenv`	return environment variable (`getenv`(3F))
`iabs`	absolute value (`abs`(3F))
`iargc`	return number of arguments (`iargc`(3F))
`ichar`	explicit type conversion (`ftype`(3F))
`idim`	positive difference (`dim`(3F))
`idint`	explicit type conversion (`ftype`(3F))
`idnint`	nearest integer (`round`(3F))
`ifix`	explicit type conversion (`ftype`(3F))
`index`	return location of substring (`index`(3F))
`int`	explicit type conversion (`ftype`(3F))
†`irand`	random number generator
`isign`	transfer of sign (`sign`(3F))
`len`	return length of string (`len`(3F))
`lge`	string comparison (`strcmp`(3F))
`lgt`	string comparison (`strcmp`(3F))
`lle`	string comparison (`strcmp`(3F))
`llt`	string comparison (`strcmp`(3F))
`log`	natural logarithm (`log`(3F))
`log10`	common logarithm (`log10`(3F))
†`lshift`	bitwise Boolean (`bool`(3F))
`max`	maximum value (`max`(3F))
`max0`	maximum value (`max`(3F))
`max1`	maximum value (`max`(3F))

†`mclock`	return Fortran time accounting (`mclock`(3F))
`min`	minimum value (`min`(3F))
`min0`	minimum value (`min`(3F))
`min1`	minimum value (`min`(3F))
`mod`	remaindering (`mod`(3F))
`nint`	nearest integer (`bool`(3F))
†`not`	bitwise Boolean (`bool`(3F))
†`or`	bitwise Boolean (`bool`(3F))
†`rand`	random number generator (`rand`(3F))
`real`	explicit type conversion (`ftype`(3F))
†`rshift`	bitwise Boolean (`bool`(3F))
`sign`	transfer of sign (`sign`(3F))
†`signal`	specify action on receipt of system signal (`signal`(3F))
`sin`	sine (`sine`(3F))
`sinh`	hyperbolic sine (`sinh`(3F))
`sngl`	explicit type conversion (`ftype`(3F))
`sqrt`	square root (`sqrt`(3F))
†`srand`	random number generator (`rand`(3F))
†`system`	issue a shell command (`system`(3F))
`tan`	tangent (`tan`(3F))
`tanh`	hyperbolic tangent (`tanh`(3F))
†`xor`	bitwise Boolean (`bool`(3F))
†`zabs`	complex absolute value (`abs`(3F))

For more information on the `f77` intrinsic function commands, see *A/UX Command Reference.*

Violations of the standard

The following sections describe the three known ways in which the A/UX system implementation of Fortran 77 violates the new American National Standard. These known exceptions are:

1. double-precision alignment
2. dummy procedure arguments
3. `t` and `tl` formats

Double-precision alignment

The Fortran 77 American National Standard permits `common` or `equivalence` statements to force a double-precision quantity onto an odd word boundary. For example,

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines require that double-precision quantities be on double word boundaries; other machines run less efficiently if this alignment rule is not observed. It is possible to tell which `equivalence` and `common` variables suffer from a forced odd alignment, but every double-precision argument must be assumed on a bad boundary.

To load a double-precision quantity on some machines, you must use two separate operations:

1. Move the upper and lower halves into the halves of an aligned temporary location.
2. Load the destination from the temporary location.

To store such a result, you must reverse the order of the above two operations.

All double-precision real and complex quantities must fall on even word boundaries on machines with corresponding hardware requirements or if the source code issues a diagnostic whenever there is a violation of the odd-boundary rule.

Dummy procedure arguments

If any argument of a procedure is of type `character`, all dummy procedure arguments of that procedure must be declared in an `external` statement. For an example illustrating this, see "Argument Lists" later in this chapter.

This requirement arises as a subtle corollary of the way Fortran represents character string arguments. A warning is printed if a dummy procedure is not declared `external`. The same code is correct (in this regard), however, if there are no `character` arguments.

`t` and `tl` formats

The `t` (absolute tab) and `tl` (leftward tab) format codes allow you to reread or rewrite part of a record that is already processed.

This compiler implementation uses "seeks." Therefore, if the standard output unit is not one that allows seeks, such as a terminal, the program is in error.

The implementation chosen includes the following benefits:

- There is no upper limit on the length of a record.
- You do not have to predeclare any record lengths, except where specifically required by Fortran or by the operating system.

Interprocedure interface

The following sections provide information necessary for writing C language procedures that call or are called by Fortran procedures. Specifically, you should understand the conventions regarding the following language concepts:

1. procedure names
2. data representation
3. return values
4. argument lists

Procedure names

On A/UX systems, the compiler appends an underscore to the name of a common block for a Fortran procedure to distinguish it from a C language procedure or an external variable with the same user-assigned name.

Fortran library procedure names have embedded underscores, to avoid clashes with user-assigned subroutine names.

Data representations

The following list shows corresponding Fortran and C language declarations:

Fortran	*C language*
`integer*2 x`	`short int x;`
`integer x`	`long int x;`
`logical x`	`long int x;`
`real x`	`float x;`
`double precision x`	`double x;`
`complex x`	`struct {float r, i;} x;`
`double complex x`	`struct {double dr, di;} x;`
`character*6 x`	`char x[6];`

By the rules of Fortran, `integer`, `logical`, and `real` data occupy the same-sized areas in memory.

Return values

A function of type `integer`, `logical`, `real`, or `double precision`, declared as a C language function, returns the corresponding type.

A `complex` or `double complex` function is equivalent to a C language routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f(arg...)
```

is equivalent to

```
struct {float r, i;} temp;
f_(&temp, arg...)
```

A **character-valued function** is equivalent to a C language routine with two extra initial arguments:

- a data address
- a length

Thus,

```
character*15 function g(arg...)
```

is equivalent to

```
char result[ ];
long int length;
g_(result, length, arg...)
```

and can be invoked in the C language by

```
char chars[15];
...
g_(chars, 15L, arg...);
```

Subroutines are invoked as if they were *integer-valued functions* whose value specifies which alternate return to use. Alternate return arguments, or **statement labels,** are not passed to the function, but are used to do an indexed branch in the calling procedure. If the subroutine has no entry points with alternate return arguments, the returned value is undefined. Thus, the statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed `goto`

```
goto (1, 2, 3), nret( )
```

Argument lists

All Fortran arguments are passed by address.

For every argument that is of type `character` or a dummy procedure, an argument giving the length of the value is passed. The string lengths are `long int` quantities passed by value.

The order of arguments is then:

1. extra arguments for complex and character functions
2. address for each datum or function
3. a `long int` for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. For example, in C, the above array of 3 elements would be subscripted 0, 1, 2; in `f77` they are subscripted 1, 2, 3.

Fortran arrays are stored in column-major order. C language arrays are stored in row-major order. The stored order for each language is given by the numbers in the sample two-dimensional arrays that follow:

```
f77:
      1 3
      2 4
```

```
C:
        1 2
        3 4
```

File formats

Fortran requires four kinds of external files:

1. sequential formatted
2. sequential unformatted
3. direct formatted
4. direct unformatted

On A/UX systems, these are all implemented as ordinary files that are assumed to have the proper internal structure.

Fortran I/O is based on **records.** When a `direct` file is opened in a Fortran program, the record length of the records must be given. This is used by the Fortran I/O system to make the file look as if it were made up of records of the given length. In the special case that the record length is given as one, the files are not considered to be divided into records but are treated as ordinary files on the A/UX system (byte-addressable byte strings). A `read` or `write` request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.

The peculiar requirements on `sequential unformatted` files make it unlikely that they are ever read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record length, in bytes.

The Fortran I/O system breaks `sequential formatted` files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined, according to the Fortran 77 American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system writes a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect is that the single record you thought was written is treated as more than one record when being read or backspaced over.

Preconnected files and file positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for `sequential formatted` I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named `fort.`*n*. These files need not exist and is not created unless their units are used without first executing an `open`. The default connection is for `sequential formatted` I/O.

The Fortran 77 Standard does not specify where a file that is opened explicitly for `sequential` I/O is positioned initially. In fact, the I/O system positions the file at the beginning. If the file contains data, a `write` overwrites it. To position a file to the end, you must read until an end-of-file condition is reached. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

13 EFL Reference

This chapter is a reference for the EFL programming language. It describes the features and use of the language; although supplemented by the chapters on Fortran, it can stand alone as an arbiter of the EFL language. To use this chapter, you should be somewhat familiar with a procedural language.

EFL is a clean, general-purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring.

EFL programs can be translated into efficient Fortran code. This means that you can take advantage of the Fortran libraries and benefit from the portability that comes with the use of a standardized language. Even though EFL originally stood for "Extended Fortran Language," the EFL compiler is much more than a simple preprocessor.

The EFL compiler attempts to diagnose all syntax errors, provide readable Fortran output, and avoid a number of Fortran restrictions. For example, while EFL allows variable white space in its input, standard Fortran requires that comment indicators and data are placed in standard, specified columns and does not compile properly if these columns are not used. In addition, EFL is a structured language, while standard Fortran uses `goto` and `continue` statements. These and other Fortran restrictions are mentioned in sections such as "Continuation Conventions" and "Miscellaneous Output Control Options," later in this chapter.

EFL is especially useful for numeric programs and lets you express complicated ideas in a comprehensible way, while giving you access to the power of the Fortran environment.

`efl` command syntax

In examples and syntax specifications in this chapter, a construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation [[*item*]] can refer to any of the following entities:

item

item, *item*

item, *item*, *item*

To increase the legibility of EFL programs, you can break some of the statement forms without an explicit continuation. A square (□) in the syntax represents a point where an end-of-line is ignored.

The A/UX `efl` command has the following syntax:

`efl` [`-w`] [`-#`] [`-C`] [*filename*...]

The options for `efl` are as follows:

`-w`	suppresses warning messages
`-#`	suppresses comments in the generated program and the flag option
`-C`	causes comments to be included in the generated program (on by default)

An argument with an embedded `=` (equal sign) sets an `efl` option as if it appeared in an `option` statement at the start of the program. Many options are described in the section "Compiler Options," later in this chapter. A set of defaults for a particular target machine can be selected by one of the choices: `system=unix`, `system=gcos`, or `system=cray`. The default setting of the `system` option is the same as the machine on which the compiler is running. Other specific options determine the style of input/output, error handling, continuation conventions, the number of characters packed per word, and default formats.

Lexical form

Character set

The characters in Table 13-1 are legal in an EFL program.

Table 13-1 Legal characters in EFL

Letters	a b c d e f g h i j k l m n o p q r s t u v w x y z
Digits	0 1 2 3 4 5 6 7 8 9
White space	blank tab
Quotes	' "
Number sign	#
Continuation	_
Braces	{ }
Parentheses	( )
Other	, ; : . + - * / = < > & ~ | $

Even though all the examples are printed in lowercase, case is ignored, except within strings (for example, `a` and `A` are treated as the same character). An exclamation mark (`!`) can be used in place of a tilde (~) as the logical unary operator "complement." Square brackets (`[` and `]`) can be used in place of braces (`{` and `}`) for punctuation.

Outside a character string or comment, a sequence of one or more spaces or tab characters acts as a single space and terminates a token.

Tokens

A program is made up of a sequence of tokens. Each **token** is a sequence of characters. A blank terminates any token except a quoted string. An end-of-line also terminates a token unless you signal explicit continuation by an underscore.

Lines

EFL is a line-oriented language. Except in special cases where continuation is made explicit by use of an underscore (_), the end of a line marks the end of a token and the end of a statement.

You can use the trailing portion of a line for a comment. Diagnostic messages are labeled with the line number of the file in which they are detected.

You can continue lines explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space are stripped) is an underscore, the end of the line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts, except inside quoted strings. Thus,

```
1_000_000_
    000
```

equals 10^9.

There are also rules for continuing lines automatically: the end-of-line is ignored whenever it's obvious that the statement is not complete. A statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis, but a statement is not continued if unbalanced braces or parentheses exist. Some compound statements also are continued automatically; these points are noted in the sections on executable statements.

Multiple statements on a line

A semicolon terminates the current statement. Therefore, you can write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

Comments

You can place a comment at the end of any line. It is introduced by a number sign (#) and continues to the end of the line. The number sign and succeeding characters on the line are discarded. A blank line is also considered a comment. Comments have no effect on execution.

◆ **Note** A number sign inside a quoted string does not mark a comment. ◆

`include` files

You can insert the contents of a file `joe` at a certain point in the source text by referencing it in the line

```
include joe
```

No statement or comment can follow an `include` on a line. In effect, the `include` line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. `include` statements can be nested at least ten deep.

Identifiers

An **identifier** is a name used in an EFL program consisting of a letter or a letter followed by letters or digits. The following list shows reserved words that have special meaning in EFL and therefore should not be used as identifiers. You should use these words only for the purposes described in this chapter.

```
array               exit          precision
automatic           external      procedure
break               false         read
call                field         readbin
case                for           real
character           function      repeat
common              go            return
complex             goto          select
continue            if            short
debug               implicit      sizeof
default             include       static
define              initial       struct
dimension           integer       subroutine
do                  internal      true
double              lengthof      until
doubleprecision     logical       value
else                long          while
end                 next          write
equivalence         option        writebin
```

Strings

A **character string** is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quotation marks (`'`), it may contain double-quotation marks (`"`), and vice versa. You may not break a quoted string across a line boundary. Legal character strings include

```
'hello there'

"ain't misbehavin'"
```

Integer constants

An **integer constant** is a sequence of one or more digits:

```
0

57

123456
```

Floating-point constants

A **floating-point constant** contains a dot, an exponent field, or both. An **exponent field** is the letter `d` or `e` followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I

I.

I.J

IE

I.E

.IE

I.JE

Punctuation

You can use certain characters to group or to separate objects in the language, as follows:

parentheses	`( )`
braces	`{ }`
comma	`,`
semicolon	`;`
colon	`:`
end-of-line	`<CR>`

The end-of-line is a token (statement separator) if the line is nonblank or noncontinued.

Operators

The EFL operators are written as sequences of one or more nonalphanumeric characters, as shown in Table 13-2.

Table 13-2 EFL operators

Operator	Meaning
`+`	unary plus (no effect)
`+`	binary plus (a + b)
`++`	prefix plus (a = a + 1)
`--`	prefix minus (a = a - 1)
`-`	binary minus (a - b)
`*`	times ($a \times b$)
`/`	divided by ($a \div b$)
`**`	exponentiation (a^b)
`<`	is less than (a < b)
`<=`	is less than or equal ($a \leq b$)
`>`	is greater than (a > b)
`>=`	is greater than or equal ($a \geq b$)
`==`	equal (a = b)

Table 13-2 EFL operators *(continued)*

Operator	Meaning
`~=`	does not equal ($a \neq b$)
`$`	repetition (2$a = aa)
`.`	fp decimal point (a.*exp field*)
`&&`	logical and ($a \wedge b$)
`||`	logical or ($a \vee b$)
`&`	and (a and b)
`|`	or (a or b)
`=`	assign equal (a "gets" b)
`+=`	assign plus ($a = a + b$)
`-=`	assign minus ($a = a - b$)
`/=`	assign divide ($a = a \div b$)
`*=`	assign times ($a = a \times b$)
`**=`	assign exp ($a = a^b$)
`&&=`	assign logical and ($a = a \wedge b$)
`||=`	assign logical or ($a = a \vee b$)
`&=`	assign and (a = a and b)
`|=`	assign or (a = a or b)
`->`	*leftside* = *structure name*

Note: "fp" stands for "floating point."

A dot (`.`) is an operator if it qualifies a structure element name, but not if it acts as a decimal point in a numeric constant. There is a special mode (see "Atavisms," later in this chapter) in which some of the operators can be represented by a string consisting of a dot, an identifier, and another dot (for example, `.lt.`).

Macros

EFL has a simple macro substitution facility. You can define an identifier to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a `define` statement such as

```
define count  n += 1
```

Any time the name `count` appears in the program, it is replaced by the statement

```
n += 1
```

A `define` statement must appear alone on a line; the format is

`define` *name definition-string*

Trailing comments are part of the *definition-string*.

Program form

Files

A **file** is a sequence of lines and is compiled as a single unit. It can contain one or more procedures. Declarations and options that appear outside a procedure affect the succeeding procedures on that file.

Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked (the first procedure invoked during execution, known as the **main** procedure, has a null name).

Block scope

You can form statements into groups inside a procedure. Then, their influence on the rest of the program is determined by their location in the program, the resulting **scope** of their effect, or both.

The beginning of a program file is at "nesting level" zero. Any options, macro definitions, or variable declarations you enter are also at level zero.

After the declarations, if you enter a left brace, this marks the beginning of a new block and increases the nesting level by one; a right brace decreases the nesting level by one. Braces that are inside declarations do not mark blocks (see "Blocks," later in this chapter, for further information on blocks).

You can then enter a procedure statement for level 1. The text immediately following the `procedure` statement is also at level 1. An `end` statement marks the end of the procedure and level 1 and returns you to level 0 within the program.

If you define a name (variable or macro) at level 0, it remains defined throughout that block and in all deeper (higher numbered: for example, 1, 2, 3) nesting levels, unless that name is redefined or redeclared. If, for example, you define a variable in level 0 (for example, `a = 7`), `a` is 7 throughout the program. If you want to include a subroutine at a deeper level and that subroutine needs `a` to equal 3, you can redefine `a` for that subroutine. `a` equals 3 in that subroutine only, however, because as soon as the program leaves the subroutine, the definition set forth in level 0 prevails.

A procedure illustrating block level scope might look like the code shown in Figure 13-1.

Figure 13-1 Procedure illustrating block level scope

```
#   block 0
procedure george
real x
x = 2
...
if(x > 2)
            {           # new block
            integer x   # a different variable
            do x = 1,7
                  write(,x)
...
}           # end of block
end     # end of procedure, return to block 0
```

Statements

Statements are of the following types:

```
option
include
define

procedure
end

declarative
executable
```

The `option` statement is described in "Compiler Options," later in this chapter. The `include`, `define`, and `end` statements were described previously; you cannot follow them with another statement on a line. Each procedure begins with a `procedure` statement and finishes with an `end` statement. Declarations or `declarative` statements describe types and values of variables and procedures. `executable` statements cause specific actions to occur. A block is an example of an executable statement; it is made up of declarative and executable statements.

Labels

An executable statement can have a label, which can be used in a branch statement. A **label** is an identifier followed by a colon, appearing at the margin to the left of some statement, such as `error:` shown here:

```
              read(, x)
              if(x < 3) goto error
              ...
error:        fatal("bad input")
```

Data types and variables

EFL supports a small number of basic (scalar) types. You can define objects made up of variables of basic type (that is, **aggregates**) and then define other defined aggregates.

Basic types

The basic types are as follows:

`logical`	A `logical` quantity can take on the two values `true` and `false`.
`integer`	An `integer` can take on any whole number value in a machine-dependent range.
`field(`*m*`:`*n*`)`	A `field` quantity is an integer restricted to a particular closed interval ([*m*:*n*]).
`real`	A `real` quantity is a floating-point approximation to a real or rational number; real quantities are represented as single-precision floating-point numbers.
`complex`	A `complex` quantity is an approximation to a complex number and is represented as a pair of `real` types.
`long real`	A `long real` is a more precise approximation to a rational number; `long real` types are double-precision floating-point numbers.

`long complex`	A `long complex` quantity is an approximation to a complex number and is represented as a pair of `long real` types.
`character(`*n*`)`	A `character` quantity is a fixed-length string of *n* characters.

Constants

There is a notation for a constant of each basic type.

A `logical` can take on the following two values:

```
true
false
```

An `integer` or `field` constant is a fixed-point constant, optionally preceded by a plus or minus sign, as in

```
17
-94
+6
0
```

A `long real` "double-precision" constant is a floating-point constant containing an exponent field that begins with the letter `d`. A `real` "single-precision" constant is any other floating-point constant. A `real` or `long real` constant can be preceded by a plus or minus sign. The following are valid `real` constants:

17.3

-.4

7.9e-6 $(= 7.9 \times 10^{-6})$

14e9 $(= 1.4 \times 10^{10})$

The following are valid `long real` constants:

7.9d-6 $(= 7.9 \times 10^{-6})$

5d3

A `character` constant is a quoted string. Consider the following example:

```
"bad input"
"I'm real, not integer"
```

Variables

A **variable** is a quantity with a name and a location; at any particular time the variable also can have a value. A variable is said to be "undefined" before it is initialized or assigned its first value.

Each variable has certain attributes:

- storage class
- scope
- precision

A variable **storage class** is the association of its name and its location. A storage class can be either transitory or permanent.

- **Transitory association** is achieved when arguments are passed to procedures.
- Other associations are considered **permanent** or **static associations.**

The scope of a variable can be global or local:

1. The names of common areas are global. **Global variables** can be used anywhere in the program, as they are known throughout the program.
2. All other names are considered **local** to the block in which they are declared.

(For further information about scope, refer to "Block Scope," earlier in this chapter.)

Floating-point variables are either of normal or long precision. **Normal precision** is 32 bits; **long precision** is 64 bits. You can state this attribute independently of the basic type.

Arrays

You can declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array can have intervals that are of length equal to that of one of the other formal arguments. An element of an array is denoted by the array name, followed by a parenthesized, comma-separated list of integer values, each of which must lie within the corresponding interval. The intervals can include negative numbers. Entire arrays can be passed as procedure arguments or in input/output lists, or they can be initialized; all other array references must be to individual elements.

For example, the declared integer array

```
array (2, 10) chance
```

can have the elements

```
chance (3)
chance (2, 8)
```

Structures

You can define new types that are made up of elements of other types. This compound object is known as a **structure;** its constituents are called **members** of the structure.

You can name the structure. This name then acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure can be of any type (including previously defined structures), or they can be arrays of such objects. You can pass entire structures to procedures or use them in input/output lists; you also can reference individual elements of structures.

The following structure might represent a symbol table:

```
struct tableentry
   {
            character(8) name
            integer hashvalue
            integer numberofelements
            field(0:1) initialized, used, set
            field(0:10) type
   }
```

Expressions

Expressions are syntactic forms that yield a value. An expression can have any of the following forms, recursively applied:

primary
(*expression*)
unary-operator expression
expression binary-operator expression

The precedence of EFL operators, pictured from highest to lowest, is shown in the following table. Lines separate the precedence levels. The meanings of these operators are described in "Unary Operators" and "Binary Operators," both later in this chapter.

Table 13-3 Precedence of operators in EFL

Priority	Operator	Meaning
Highest	->	leftside = structure name
	.	fp decimal point (a.exp field)
	**	exponentiation (a^b)
	*	times ($a \times b$)
	/	divided by ($a \div b$)
	+	unary plus (no effect)
	-	unary minus (negation)
	++	prefix plus (a = a + 1)
	--	prefix minus (a = a - 1)
	+	binary plus (a + b)
	-	binary minus (a - b)
	<	is less than (a < b)
	<=	is less than or equal ($a \leq b$)
	>	is greater than (a > b)
	>=	is greater than or equal ($a \geq b$)
	==	equal ($a \equiv b$)
	~=	does not equal ($a \neq b$)
	&	and (a and b)
	&&	logical and ($a \wedge b$)

(continued)➡

Table 13-3 Precedence of operators in EFL *(continued)*

Priority	Operator	Meaning
	`|`	or (a or b)
	`||`	logical or (a ∨ b)
	`$`	repetition (2$a = aa)
Lowest	`=`	assign equal (a "gets" b)
	`+=`	assign plus (a = a + b)
	`-=`	assign minus (a = a - b)
	`*=`	assign times (a =a × b)
	`/=`	assign divide (a = a ÷b)
	`**=`	assign exp (a = a^b)
	`&=`	assign and (a = a and b)
	`|=`	assign or (a = a or b)
	`&&=`	assign logical and (a = a ∧ b)
	`||=`	assign logical or (a = a ⊥ b)

The following are examples of expressions:

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

Constants

Constants are described in "Constants," earlier in this chapter.

Variables

Scalar variable names are primaries. They can appear on the left or right side of an assignment. Unqualified names of aggregates (structures or arrays) can appear only as procedure arguments and in input/output lists.

Array elements

You can denote an element of an array with the array name, followed by a parenthesized list of subscripts, with one integer value for each declared dimension.

```
a(5)
b(6,-3,4)
```

Structure members

A structure name, followed by a dot, followed by the name of a member of that structure constitutes a **reference** to that element. If that element is itself a structure, the reference can be further qualified.

```
a.b
x(3).y(4).z(5)
```

Procedure invocations

You can invoke a procedure by an expression of one of the forms

procedurename `( )`
procedurename `(`*expression*`)`
procedurename `(`*expression-1*`, ...,` *expression-n*`)`

The *procedurename* is either the name of a variable declared `external` (see "Attributes," later in this chapter), the name of a function known to the EFL compiler (see "Known Functions," later in this chapter), or the actual name of a procedure as it appears in a `procedure` statement. If a *procedurename* is declared external and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise, it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*.

The following examples are procedure invocations:

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is going to be performed, each of the actual argument expressions is evaluated first. The types, precisions, and bounds of actual and formal arguments should agree.

If an actual argument is a variable name, array element, or structure member, the called procedure can use the corresponding formal argument as the left side of an assignment or in an input list; otherwise, it can use only the value.

After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a `return` statement is executed in that procedure, or when control reaches the `end` statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure. These must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument (see "Procedures," later in this chapter).

Input/output expressions

The EFL input/output syntactic forms can be used as integer primaries that have a nonzero value, if an error occurs during the input or output.

Coercions

You can **coerce** or convert an expression of one precision or type to another by an expression with the form

attributes (*expression*)

At present, the only attributes permitted are precision and basic types. Attributes are separated by white space.

An arithmetic value of one type can be coerced to any other arithmetic type. A character expression of one length can be coerced to a character expression of another length. Logical expressions cannot be coerced to a nonlogical type.

As a special case, a quantity of `complex` or `long complex` type can be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

```
integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = 5+3i
```

Most conversions are done implicitly, as most binary operators permit operands of different arithmetic types. Explicit coercions are most useful when you need to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

Sizes

The notation that yields the amount of memory required to store a datum or an item of specified type is

`sizeof (`*leftside*`)`

`sizeof (`*attributes*`)`

In the first case, *leftside* can denote a variable, array, array element, or structure member. In the second case, *attributes* can denote an item of a specified type. The value of `sizeof` is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

```
sizeof(x) / sizeof(integer)
```

yields the size of the variable `x` in integer words.

The distance between consecutive elements of an array cannot equal `sizeof` because certain data types require final padding on some machines. The `lengthof` operator gives this larger value, again in arbitrary units. The syntax is as follows:

`lengthof (`*leftside*`)`

`lengthof (`*attributes*`)`

Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression is evaluated before any larger expression of which it is a part is evaluated.

Unary operators

All the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

Arithmetic

Unary `+` has no effect. A unary `-` yields the negative of its operand.

The prefix operator `++` adds one to its operand. The prefix operator `--` subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. As a side effect, the operand value is changed.

Logical

The only logical unary operator is complement (~). This operator is defined by the following equations.

```
~ true = false
~ false = true
```

Binary operators

Most EFL operators have two operands separated by the operator. Because the character set is limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

Arithmetic

The binary arithmetic operators are as follows:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation

Exponentiation is right associative:

```
a**b**c = a**(b**c) = a^(bc)
```

The operations have the conventional meanings:

```
8 + 2 = 10

8 - 2 = 6

8 * 2 = 16

8/2 = 4

8 ** 2 = 8^2 = 64
```

The type of the result of a binary operation `A` *op* `B` is determined by the types of its operands, as shown in Table 13-4.

Table 13-4 Type of result of binary operation `A` *op* `B`

	Type of B				
Type of A	**i**	**r**	**lr**	**c**	**lc**
i	i	r	lr	c	lc
r	r	r	lr	c	lc
lr	lr	lr	lr	lc	lc
c	c	c	lc	c	lc
lc	lc	lc	lc	lc	lc

Note: **i** = integer, **r** = real, **lr** = long real, **c** = complex, **lc** = long complex.

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer and is computed exactly (quotients are truncated toward zero, so 8/3 = 2).

Logical

The two binary logical operations in EFL, `and` and `or`, are defined by the truth tables shown in Table 13-5.

Table 13-5 Truth tables for `and` and `or`

A	B	A **and** B	A **or** B
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

```
a && b
```

is evaluated by first evaluating `a`; if it is false, the expression is false and `b` is not evaluated; otherwise, the expression has the value of `b`. The expression

```
a || b
```

is evaluated by first evaluating `a`; if it is true, then the expression is true and `b` is not evaluated; otherwise, the expression has the value of `b`. The other forms of the operators (`&` for `and`, and `|` for `or`) do not imply an order of evaluation. With the latter operators, the compiler can evaluate the operands in any order, thus speeding up the code.

Relational operators

There are six relations between arithmetic quantities. These operators are not associative.

Table 13-6 Relational operators in EFL

EFL operator	**Meaning**
`<`	< Less than
`<=`	≤ Less than or equal to
`==`	= Equal to
`~=`	≠ Not equal to
`>`	> Greater than
`>=`	≥ Greater than or equal to

Because the complex numbers are not ordered, the only relational operators that can take complex operands are == and ~=. The character collating sequence is not defined.

Assignment operators

All the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

Corresponding to each binary operator is an assignment operator. For each binary operator, the assignment operator is formed by concatenating an equal sign (=) to the operator *with no space between them*. For the case of binary +, the assignment operator becomes +=, and the assignment

```
a += b
```

is translated as

```
a = a + b
```

Thus, the assignment

```
n += 2
```

adds 2 to n. The *basic-left-side* is evaluated only once.

Dynamic structures

EFL does not have an address (pointer, reference) type. There is a notation, however, for dynamic structures:

left-side -> *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *left-side*. In effect, this overlays the structure template on the specified location. The *left-side* must be a variable, array, array element, or structure member. The type of the *left-side* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way, using the dot operator. Thus,

```
place(i) -> st.nth
```

refers to the `nth` member of the `st` structure starting at the `i`th element of the array `place`.

Repetition operator

Inside a list, an element of the form

integer-constant-expression `$` *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

```
(3, 3$4, 5)
```

is equivalent to

```
(3, 4, 4, 4, 5)
```

Constant expressions

If you build an expression out of operators (other than functions) and constants, the value of the expression is a constant and can be used anywhere a constant is required.

Declarations

Declaration statements describe the meaning, shape, and size of named objects in the EFL language.

Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two forms:

attributes variable-list
attributes { *declarations*}

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the *declarations* also has the specified attributes. A variable name may appear in more than one variable list, as long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Here are some examples of declarations:

```
integer k=2

long real b(7,3)

common(cname)
            {
            integer i
            long real array(5,0:3) x, y
            character(7) ch
            }
```

Attributes

The following attributes are basic types in declarations:

```
logical
integer
field(m:n)
character(k)
real
complex
```

In the above list, the quantities *k*, *m*, and *n* denote integer constant expressions with the properties $k > 0$ and $n > m$.

Arrays

The dimensionality can be declared by an `array` attribute:

`array(` $b_1, \ldots, b_n$ `)`

Each of the b_i can be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to *n*, the number of bounds.

Each of the integer expressions must be a constant. An exception is permitted only if each of the variables associated with an array declarator is a formal argument of the procedure. In this case, each bound must have the property that *upper* - *lower* + *1* is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it recognizes important cases such as `(0:n-1)`.) The upper bound for the last dimension (b_n) can be marked by an asterisk (`*`) if the size of the array is unknown.

The following attributes are legal `array`:

```
array(5)

array(5, 1:5, -3:0)

array(5, *)

array(0:m-1, m)
```

Structures

A structure declaration is of the form

`struct [`*structname*`] {`*declarations*`}`

If the optional *structname* is present, it takes the place of a type name within the rest of its scope. Each name that appears inside a *declaration* is a **member** of the structure and has a special meaning when used to qualify any variable declared with the structure type. The *declarations* inside the braces are one or more declaration statements.

A name can appear as a member of any number of structures. It also can be the name of an ordinary variable, as a structure member name is used only in contexts where the parent type is known.

The following code shows examples of valid structure attributes:

```
struct xx
            {
            integer a, b
            real x(5)
            }
struct { xx z(3); character(5) y }
```

The last line defines a structure that contains an array of three `xx`s and a character string.

Precision

Variables of floating-point (`real` or `complex`) type can be declared to be `long` to ensure that they have higher precision than ordinary floating-point variables. The default precision is `short`.

Common

Certain objects called *common areas* have external scope and can be referenced by any procedure that has a declaration for the name using a

`common (`*common-area-name*`)`

attribute. All the variables declared with a particular `common` attribute are in the same block. The order in which they are declared is significant; declarations for the same block in different procedures must have the variables in the same order and with the same types, precision, and shapes, although not necessarily with the same names.

External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the `external` attribute. If a procedure name is to be passed as an argument, you must declare it in a statement with the form

external [[*name*]]

If a name has the `external` attribute and is a formal argument of the procedure, it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, it is the actual name of a procedure as it appears in the corresponding procedure statement.

Variable list

The variable list in a declaration consists of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules.

The dimension specification has the same form and meaning as the list enclosed in parentheses in an array attribute.

The initial value specification has an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign can be a list of constant expressions or repeated elements or lists enclosed in parentheses; the total number of elements in the list must not exceed the number of elements in the array. Array elements are filled in column-major order.

The initial statement

An initial value also can be specified for a simple variable, array, array element, or member of a structure using a statement with the form

initial [[*var* = *val*]]

where *var* can be a variable name, array element specification, or member of structure, and *val* is the initial value specified.

The right side follows the same rules as for an initial value specification in other declaration statements.

Executable statements

Every useful EFL program must contain executable statements; otherwise, it cannot do anything. Executable statements are frequently made up of other statements. While blocks are the most obvious example of this, many other forms are made up of statements as well.

To increase the legibility of EFL programs, you can break some of the statement forms without an explicit continuation. A square (□) in the syntax represents a point where an end-of-line is ignored.

Expression statements

A procedure invocation that returns no value is known as a **subroutine call.** Such an invocation is a statement. Examples include:

```
work(in, out)
run()
```

Input/output statements (see “Input/Output Statements,” later in this chapter) resemble procedure invocations but do not yield a value. If an error occurs here, the program stops.

An expression that is a simple assignment (=) or a compound assignment (+=, -=, and so on) is a statement, such as

```
a = b
a = sin(x)/6
x *= y
```

Blocks

A **block** is a compound statement that acts as a single statement. A block uses the following syntax:

{ [[*declaration*]] [[*executable-statement*]] }

A block can be used anywhere a statement is permitted. A block is not an expression and does not have a value. The following code shows a sample block.

```
{
integer i  # this variable is unknown
           # outside the braces of this block
big = 0
do i = 1,n
    if(big < a(i))
           big = a(i)
}
```

Test statements

A **test statement** permits execution of another statement or group of statements based on the outcome of a conditional expression.

There are several forms of test statements:

- `if` statements
- `if-else` statements
- `select` statements

`if` statement

The simplest of the test statements is the `if` statement. Its form is

`if (`*logical-expression*`)` ☐ *statement*

where ☐ means the line can be broken at this point.

First, the *logical-expression* is evaluated; if it is true, the *statement* is executed; if it is not, the *statement* is skipped.

`if-else` statement

A more general statement is of the form

`if (`*logical-expression*`)` ☐ *statement-1* ☐
`else` ☐ *statement-2*

where ☐ means the line can be broken at this point.

Just as with the `if` statement, the *logical-expression* is evaluated and, if it's true, *statement-1* is executed; if not, *statement-2* is executed. Either of the consequent statements can itself be an `if-else` statement, so a completely nested test sequence is possible. For example,

```
if(x<y)
          if(a<b)
                    k = 1
          else
                    k = 2
else
          if(a<b)
                    m = 1
          else
                    m = 2
```

An `else` statement applies to the nearest preceding `if` that is not already followed by an `else`.

A more common use of the `if-else` test statement is the sequential test:

```
if(x==1)
   k = 1
else if(x==3  |   x==5)
   k = 2
else
   k = 3
```

You can use any number of `else if` statements within a single `if-else` statement to test for several conditions; if, however, you need more than two `else if` statements, you might prefer to use a select statement instead.

`select` statement

Much like the `switch` statement in the C shell or `case` statements in many programming languages, a `select` **statement** is used to direct the branching of a program based on the result of a conditional or arithmetic expression. A `select` statement has the general form

`select(`*expression*`)` □ *block*

Inside the block, two special types of labels are recognized. A prefix with the form

`case [[`*constant*`]]:`

marks the statement to which control is passed if the value of the expression in the `select` is equal to one of the `case` constants. If the expression does not equal any of these constants but there is a label `default` inside the `select`, a branch is taken to that point; otherwise, the statement following the right brace is executed.

Once execution begins at a `case` or `default` label, it continues until the next `case` or `default` is encountered. An example follows:

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Loops

The **loop constructs** (`while`, `for`, `repeat`, `repeat-until`, and `do`) provide an efficient way to repeat an operation or series of operations. Loop termination is generally initiated by the failure of a logical or iterative test statement. Although the `while` loop is the simplest construct, and consequently the most frequently used, each construct has its own strengths to be exploited in a given application.

`while` statement

This construct has the form

`while (`*logical-expression*`)` □ *statement*

First, the *logical-expression* is evaluated; if it is true, *statement* is executed and the *logical-expression* is evaluated again. If it is false, *statement* is not executed and program execution continues at the next statement.

`for` statement

The `for` statement is a more elaborate looping construct. It has the form

`for (`*initial-statement*`,` □ *logical-expression*`,`
□ *iteration-statement*`)` □ *body-statement*

Except for the behavior of the `next` statement (see "Branch Statements," later in this chapter), this construct is equivalent to

initial-statement

`while (`*logical-expression*`)`
    `{`
    body-statement
    iteration-statement
    `}`

This form is useful for general arithmetic iterations and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
        n += i
```

Alternatively, the computation can be done by the single statement

```
for({n=0; i=1}, i<=100, {n+=i; ++i})
        ;
```

Note that the body of the `for` loop is a null statement in this case. An example of following a linked list is provided later.

`repeat` statement

The statement

`repeat` □ *statement*

executes the *statement*, then executes it again, without any termination test. A test inside the *statement* is needed to stop the loop.

`repeat-until` statement

The `while` loop performs a test before each iteration. The statement

`repeat` □ *statement* □ `until` `(`*logical-expression*`)`

executes the *statement*, then evaluates the *logical-expression*. If the *logical-expression* is true, the loop is complete; otherwise, control returns to the *statement*. Thus, the body is always executed at least once. The `until` refers to the nearest preceding `repeat` that is not paired with an `until`. In practice, this appears to be the least frequently used looping construct.

`do` loop

The simple arithmetic progression is a very common one in numeric applications. EFL has a special loop form for ranging over an ascending arithmetic sequence:

`do` *variable* `=` *expression-1*`,` *expression-2*`,` *expression-3*
    statement

The variable is first given the value *expression-1*. The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2*. If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

`t2 =` *expression-2*
`t3 =` *expression-3*
`for(`*variable*`=`*expression-1*`,` *variable*`<=`*t2*`,` *variable*`+=`*t3*`)`
    statement

(the compiler translates EFL `do` statements into Fortran `do` statements, which are usually compiled into excellent code). The `do` *variable* cannot be changed inside of the loop and *expression-1* must not exceed *expression-2*. The sum of the first hundred positive integers can be computed by the following code:

```
n = 0
do i = 1, 100
            n += i
```

Branch statements

It is not considered good programming practice to use branch statements if you can use a loop construct instead. If you must use a branch statement, however, EFL provides a few for your convenience.

`goto` **statement**

The most general, and most risky, branch statement is the simple, unconditional

`goto` *label*

After this statement, the statement following the given label is performed. Inside a `select`, the `case` labels of that block can be used as labels, for example,

```
select(k)
     {
     case 1:
          error(7)
           case 2:
                      k = 2
                      goto case 4
     case 3:
          k = 5
          goto case 4
     case 4:
          fixup(k)
          goto default
     default:
          prmsg("ouch")
     }
```

If two `select` statements are nested, the `case` labels of the outer `select` are not accessible from the inner one.

`break` statement

A safer statement is one that transfers control to the statement following the current `select` or `loop` form. A statement of this sort is almost always needed in a `repeat` loop:

```
repeat
    {
    do a computation
    if (finished)
    break
    }
```

More general forms permit controlling a branch out of more than one construct. For example,

```
break 3
```

transfers control to the statement following the third loop and `select` surrounding the statement.

You can specify the type of construct from which control is to be transferred, for example, `for`, `while`, `repeat`, `do`, or `select`. For example,

```
break while
```

breaks out of the first surrounding `while` statement. Either of the statements

```
break 3 for
break for 3
```

transfers to the statement after the third enclosing `for` loop.

`next` statement

The `next` statement causes the first surrounding loop statement to go to the next iteration. The next operation performed is the test of a `while`, the iteration-statement of a `for`, the body of a `repeat`, the test of a `repeat...until`, or the increment of a `do`. Elaborations similar to those for `break` are available, as follows.

```
next
next 3
next 3 for
next for 3
```

A `next` statement ignores `select` statements.

`return` statement

The last statement of a procedure is followed by a return of control to the caller. If you want to effect such a return from any other point in the procedure, a `return` statement should be executed. Inside a function procedure, the function value is specified as an argument of the statement

`return (`*expression*`)`

Input/output statements

EFL has two input statements (`read` and `readbin`), two output statements (`write` and `writebin`), and three control statements (`endfile`, `rewind`, and `backspace`). You can use any of these forms either as a primary with an `integer` value or as a statement.

If an exception occurs when one of these forms is used as a statement, the result is undefined but probably treated as a fatal error. If these forms are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value indicates an error. EFL input/output statements reflect very strongly the facilities of Fortran.

I/O units

Each I/O statement refers to a "unit," which is identified by a small positive integer. Two special units are defined by EFL: the "standard input unit" and the "standard output unit." If no unit is specified in an I/O transmission statement, these units are assumed.

The data on the unit are organized into **records.** These records can be read or written in a fixed sequence. Each transmission moves an integral number of records. Transmission proceeds from the first record until the end-of-file character is reached.

Binary I/O

The `readbin` and `writebin` statements transmit data in a machine-dependent but swift manner. The statements are of the form

`writebin(`*unit*`,` *binary-output-list*`)`

`readbin(`*unit*`,` *binary-input-list*`)`

Each statement moves one unformatted record between storage and the device. *unit* is an integer expression. A *binary-output-list* is an *iolist* without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers, in which each of the expressions is a variable name, array element, or structure member.

Formatted I/O

The `read` and `write` statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

`write(`*unit*`,` *formatted-output-list*`)`

`read(`*unit*`,` *formatted-input-list*`)`

The lists are of the same form as for binary I/O, except that they can include format specifications. If *unit* is omitted, the standard input or output unit is used.

Iolists

An iolist specifies a set of values to be written or a set of variables into which values are to be read. An **iolist** is a list of one or more *ioexpressions,* with the form

expression
`{`*iolist*`}`
do-specification `{`*iolist*`}`

For formatted I/O, an **ioexpression** also can have the forms

ioexpression`:` *format-specifier*
`:` *format-specifier*

A *do-specification* looks just like a `d` statement and has a similar effect: the values in the braces are transmitted repeatedly until the `do` execution is complete.

Formats

The following formats are permissible *format-specifiers*. The quantities *w*, *d*, and *k* must be integer constant expressions:

`i(`*w*`)`	integer with *w* digits
`f(`*w*, *d*`)`	floating-point number of *w* characters, *d* of them to the right of the decimal point
`e(`*w*, *d*`)`	floating-point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter `e`
`l(`*w*`)`	logical field of width *w* characters, the first of which is `t` or `f` (the rest are blank on output, ignored on input), standing for `true` and `false`, respectively
`c`	character string of width equal to the length of the datum
`c(`*w*`)`	character string of width *w*
`s(`*k*`)`	skip *k* lines
`x(`*k*`)`	skip *k* spaces
...	use the characters inside the string as a Fortran format

If you do not specify a format for an item in a formatted input/output statement, the EFL compiler chooses a default form.

If an item in a list is an array name, the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order that is used for array initializations.

Manipulation statements

The three input/output statements

`backspace(`*unit*`)`

`rewind(`*unit*`)`

`endfile(`*unit*`)`

look like ordinary procedure calls, but you can use them either as statements or as integer expressions that yield nonzero if an error is detected.

`backspace` causes the specified *unit* to back up so that the next `read` rereads the previous record and the next `write` overwrites it.

`rewind` moves the device to its beginning so that the next input statement reads the first record.

`endfile` causes the file to be marked so that the record most recently written is the last record on the file and any attempt to read past it is an error.

Procedures

Procedures are the basic unit of an EFL program and provide the means of segmenting a program into separately compilable and named parts.

`procedure` statement

Each procedure begins with a statement with one of the following forms:

`procedure`
attributes `procedure` *procedurename*
attributes `procedure` *procedurename* `( )`
attributes `procedure` *procedurename* `(`[[*name*]]`)`

The first form specifies the main procedure, where execution begins. In the other forms, the *attributes* can specify precision and type or they can be omitted entirely. You can declare the procedure precision and type in an ordinary declaration statement. If you do not declare a type, the procedure is a subroutine and no value can be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call.

Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

`end` statement

Each procedure terminates with the statement

`end`

Argument association

When a procedure is invoked, the actual arguments are evaluated. If the actual argument is one of the following:

- the name of a variable
- an array element
- a structure member

that entity becomes associated with the formal argument. The procedure can reference the values in the entity and assign values to it. Otherwise, the value of the actual argument is associated with the formal argument, but the procedure cannot change the formal argument value.

If the value of one of the arguments is changed in the procedure, the corresponding actual argument is not permitted to be associated with another formal argument or with a common element that is referenced in the procedure.

Execution and return values

After actual and formal arguments are associated, control passes to the first executable statement of the procedure. Control returns to the invoker when the `end` statement of the procedure is reached or when a `return` statement is executed. If the procedure is a function (has a declared type) and a `return(` *value* `)` is executed, the *value* is coerced to the correct type and precision and returned.

Known functions

A number of functions that are known to EFL need not be declared. The compiler knows the types of these functions. Some of them are generic; that is, they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke, based on the attributes of the actual arguments.

Minimum and maximum functions

The generic functions are `min` and `max`. The `min` calls return the value of their smallest argument; the `max` calls return the value of their largest argument. These are the only functions that can take different numbers of arguments in different calls. If any of the arguments are `long real`, then the result is `long real`. If any of the arguments are `real`, the result is `real`. Otherwise, all arguments and results must be `integer`. Sample function calls follow:

```
min(5, x, -3.20)
```

```
max(i, z)
```

Absolute value

The `abs` function is a generic function that returns the magnitude of its argument. For `integer` and `real` arguments, the type of the result is identical to the type of the argument; for `complex` arguments, the type of the result is the `real` of the same precision.

Elementary functions

Generic functions take arguments of `real`, `long real`, or `complex` type and return a result of the same type:

Function	*Description*
`sin`	sine function
`cos`	cosine function
`exp`	exponential function (e^x)
`log`	natural (base *e*) logarithm
`log10`	common (base 10) logarithm
`sqrt`	square root function ($\sqrt{x}$)

In addition, the following functions accept only `real` or `long real` arguments:

Function	*Description*
`atan`	arctangent function
`atan2`	arctangent of x/y

Other generic functions

The `sign` function takes two arguments of identical type: *x* and *y*. It returns positive *x* or negative *x* according to the sign of *y*.

The `mod` function yields the remainder of its first argument divided by its second argument:

Function	*Description*
`sign(`*x*`,`*y*`)`	sign conversion function
`mod(`*x*`,`*y*`)`	remainder function

These functions accept integer and real arguments.

Atavisms

The following constructs are included to ease the conversion of old Fortran programs to EFL.

Escape lines

To make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. Such a line is called an **escape line** and must begin with a percent sign (`%`). Escape lines are copied through to the output without change, except that the percent sign is removed. Inside a procedure, each escape line is treated as an executable statement. If a sequence of lines constitutes a continued Fortran statement, you should enclose it in braces.

`call` statement

You can precede a subroutine call with the keyword `call`, as follows:

```
call joe

call work(17)
```

Obsolete keywords

The keywords in Table 13-7 are recognized as synonyms of EFL keywords.

Table 13-7 Recognized keyword synonyms

Fortran	**EFL**
`double precision`	`long real`
`function`	`procedure`
`subroutine`	`procedure (untyped)`

Numeric labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted. The colon is optional following a numeric label.

Implicit declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise, it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types can be given in an `implicit` statement, with syntax

`implicit (` *letter-list* `)` *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no `implicit` statement appears, the following rules are assumed:

```
implicit (a-h, o-z) real

implicit (i-n) integer
```

Computed `goto`

Fortran contains an indexed multiway branch. You can use this facility in EFL by the computed `goto`

`goto ([[` *label* `]]),` *expression*

The expression must be of type `integer` and positive, but no larger than the number of labels in the list. Control is passed to the statement that is marked by the label whose position in the list is equal to the expression.

`goto` statement

In unconditional and computed `goto` statements, you can separate the `go` and `to` words, as in

`go to` *xyz*

Dot names

Fortran uses a restricted character set and represents certain operators (*op*) by multicharacter sequences. There is an option, `dots=on` (see "Compiler Options," later in this chapter), that forces the compiler to recognize the forms in the second column in Table 13-8.

Table 13-8 Regular and `dots=on` forms of operators

EFL op	`dots=on` **form**
`<`	`.lt.`
`<=`	`.le.`
`>`	`.gt.`
`>=`	`.ge.`
`==`	`.eq.`
`~=`	`.ne.`
`&`	`.and.`
`|`	`.or.`
`&&`	`.andand.`
`||`	`.oror.`
`~`	`.not.`
`true`	`.true.`
`false`	`.false.`

In this mode, you cannot name any structure element `lt`, `le`, and so on. The basic forms in the left column, however, are always recognized.

Complex constants

You can write a complex constant as a list of real quantities enclosed in parentheses, such as

```
(1.5, 3.0)
```

The preferred notation is by type coercion, as follows:

```
complex(1.5, 3.0)
```

Function values

The preferred way to return a value from a function in EFL is the `return (`*value*`)` construct. The name of the function acts as a variable to which values can be assigned; however, an ordinary `return` statement returns the last value assigned to that name as the function value.

Equivalence

A statement with the form

`equivalence` v_1 , v_2, ..., v_n

declares that each of the v_i starts at the same memory location. Each of the v_i can be a variable name, array element name, or structure member.

Minimum and maximum functions

There are a number of nongeneric functions in this category that differ in the types of arguments they require and types of return values. They also can have variable numbers of arguments, but all the arguments must have the same type. The nongeneric functions are shown in Table 13-9.

Table 13-9 Nongeneric functions

Function	**Argument type**	**Result type**
`amin0`	`integer`	`real`
`amin1`	`real`	`real`
`min0`	`integer`	`integer`
`min1`	`real`	`integer`
`dmin1`	`long real`	`long real`
`amax0`	`integer`	`real`
`amax1`	`real`	`real`
`max0`	`integer`	`integer`
`max1`	`real`	`integer`
`dmax1`	`long real`	`long real`

Compiler options

You can use a number of options to control the output and tailor it for various compilers and systems. The chosen defaults are conservative, but you might sometimes need to change the output to match peculiarities of the target environment.

Options are set with statements with the form

`option` [[*opt*]]

where each *opt* is of one of the forms

optionname

optionname=*optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names `yes` and `no` apply to a number of options.

Default options

Each option has a default setting. You can change the whole set of defaults to those appropriate for a particular environment by using the `system` option. At present, the only valid values are `system=unix` and `system=gcos`.

Input language options

The `dots` option determines whether the compiler recognizes `.lt.` and similar forms. The default setting is `no`.

Input/output error handling

The `ioerror` option can be given three values: `none`, `ibm`, or `fortran77`. The `none` value means that none of the I/O statements can be used in expressions, as there is no way to detect errors. The implementation of the `ibm` form uses `ERR=` and `END=` clauses. The implementation of the `fortran77` form uses `IOSTAT=` clauses.

Continuation conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option `continue=column1` puts an ampersand (`&`) in the first column of the continued lines instead.

Default formats

If you do not specify a format for a datum in an iolist for a `read` or `write` statement, a default is provided. You can change the default formats by setting certain options, as shown in Table 13-10.

The associated value must be a Fortran format, such as

```
option rformat=2.6
```

Table 13-10 Options for changing default `read/write` formats

Option	Type
`iformat`	`integer`
`rformat`	`real`
`dformat`	`long real`
`zformat`	`complex`
`zdformat`	`long complex`
`lformat`	`logical`

Alignments and sizes

To implement character variables, structures, and the `sizeof` and `lengthof` operators, you must know how much space various Fortran data types require and what boundary alignment properties they demand. The relevant options are shown in Table 13-11.

The sizes are in terms of an arbitrary unit; the alignment is in the same unit. The option `charprint` gives the number of characters per `integer` variable.

Table 13-11 Alignment and size options for Fortran data types

Fortran type	Size option	Alignment option
`integer`	`isize`	`ialign`
`real`	`rsize`	`ralign`
`long real`	`dsize`	`dalign`
`complex`	`zsize`	`zalign`
`logical`	`lsize`	`lalign`

Default input/output units

The options `ftnin` and `ftnout` are the numbers of the standard input and output units. The default values are `ftnin=5` and `ftnout=6`.

Miscellaneous output control options

Each Fortran procedure the compiler generates is preceded by the value of the `procheader` option.

No Hollerith strings are passed as subroutine arguments if `hollincall=no` is specified.

The Fortran statement numbers normally start at one and increase by one. You can change the increment value by using the `deltastno` option.

Examples

The following short examples of EFL programming show some of the convenience of the language.

File copying

The short program in Figure 13-2 copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

Figure 13-2 File-copying example

```
procedure   # main program
character(100) line

while(read(, line) == 0)
    write(, line)
end
```

Because `read` returns zero until the end-of-file (or a `read` error), this program keeps reading and writing until the input is exhausted.

Matrix multiplication

The procedure in Figure 13-3 multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula.

Figure 13-3 Matrix multiplication example

$c_{ij} = \Sigma\, a_{ik}\, b_{kj}$

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)

do i = 1,m
do j = 1,p
    {
    c(i,j) = 0
    do k = 1,n
        c(i,j) += a(i,k) * b(k,j)
    }
end
```

Searching a linked list

If you have a list of number pairs (x, y), that list is stored as a linked list and sorted in ascending order of `x` values. The procedure in Figure 13-4 searches this list for a particular value of x and returns the corresponding y value.

Figure 13-4 Example of searching a linked list

```
define LAST     0
define NOTFOUND    -1

integer procedure val(list, first, x)

#   list is an array of structures.
#   Each structure contains a thread index value,
#   an x, and a y value.

struct
    {
    integer nextindex
    integer x, y
    } list(*)

integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x ,
    p = list(p).nextindex)
    if(list(p).x == x)
        return( list(p).y )

return(NOTFOUND)
end
```

The search is a single `for` loop that begins with the head of the list and examines items until the list is exhausted (`p==LAST`) or it is known that the specified value is not on the list (`list(p).x > x`). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the `list(p)` reference. Therefore, the `&&` operator is used. The next element in the chain is found by the iteration statement `p=list(p).nextindex`.

Walking a tree

An example of a more complicated problem is if you have an expression tree stored in a common area and you want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or a binary operator, pointing to a left and right descendent. In a recursive language, such a tree walk is implemented by the simple pseudocode shown in Figure 13-5.

Figure 13-5 Pseudocode for a tree walk

```
if this node is a leaf
   print its value
otherwise
   print a left parenthesis
   print the left node
   print the operator
   print the right node
   print a right parenthesis
```

In a nonrecursive language like EFL, you need to maintain an explicit stack to keep track of the current state of the computation. The procedure in Figure 13-6 calls a procedure `outch` to print a single character and a procedure `outval` to print a value.

Figure 13-6 Example of walking a tree

```
procedure walk(first)    # print an expression tree

integer first    # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
    {
    character(1) op
    integer leftp, rightp
    real val
    } tree(100)    # array of structures
```

```
struct
    {
    integer nextstate
    integer nodep
    } stackframe(100)

define NODE    tree(currentnode)
define STACK   stackframe(stackdepth)

#   nextstate values
define DOWN    1
define LEFT    2
define RIGHT   3

#  initialize stack with root mode
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

while(stackdepth > 0)
    {
    currentnode = STACK.nodep
    select(STACK.nextstate)
        {
        case DOWN:
            if(NODE.op == " ") # a leaf
                {
                outval(NODE.val)
                stackdepth -= 1
                }
```

(continued)➨

```
                    else {# a binary operator node
                        outch("(")
                        STACK.nextstate = LEFT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.leftp
                        }
                case LEFT:
                    outch(NODE.op)
                    STACK.nextstate = RIGHT
                    stackdepth += 1
                    STACK.nextstate = DOWN
                    STACK.nodep = NODE.rightp
                case RIGHT:
                    outch(")")
                    stackdepth -= 1
                }
        }
    end
```

Portability

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the `fortran77` option is specified).

Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

Character string copying

Call the subroutine `eflasc` to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

It must copy the first `lb` characters from `b` to the first `la` characters of `a`.

Character string comparisons

The function `eflcmc` is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if string `a` of length `la` precedes string `b` of length `lb`. It returns zero if the strings are equal and a positive value otherwise. If the strings are of different lengths, the comparison is carried out as if the end of the shorter string were padded with blanks.

Compiler

Current version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all the features of the language described earlier except for `long complex` numbers.

Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and filename (if known) in which the error was detected. Warnings are given for variables that are used but not explicitly declared.

Quality of Fortran produced

The Fortran produced by EFL is clean and readable. The variable names that appear in the EFL program are used in the Fortran code when possible, and the bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded `goto` and `continue` statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). Figure 13-7 shows the Fortran procedure produced by the EFL compiler for the matrix multiplication example (see "Examples," earlier in this chapter).

Figure 13-7 Fortran code produced from matrix multiplication example

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
         do  2 j = 1, p
            c(i, j) = 0
            do 1 k = 1, n
              c(i, j) = c(i, j)+a(i, k)*b(k, j)
1           continue
2        continue
3     continue
      end
```

Figure 13-8 shows the procedure for the tree walk.

Figure 13-8 Fortran code produced from tree-walk example

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
    1 if (stapth .le. 0) goto 9
         curode = staame(2, stapth)
         goto 7
    2 if (tree(1, curode) .ne. const1(1)) goto 3
              call outval(tree1(4, curode))
c a leaf
              stapth = stapth-1
              goto 4
    3 call outch(1h()
```

(continued)➡

```
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
   4         goto 8
   5         call outch(tree(1, curode))
             staame(1, stapth) = 3
             stapth = stapth+1
             staame(1, stapth) = 1
             staame(2, stapth) = tree(3, curode)
             goto 8
   6         call outch(1h))
             stapth = stapth-1
             goto 8
   7         if (staame(1, stapth) .eq. 3) goto 6
             if (staame(1, stapth) .eq. 2) goto 5
             if (staame(1, stapth) .eq. 1) goto 2
   8      continue
          goto 1
9      continue
       end
```

Constraints on EFL

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. Implementation strategy constrained the design of EFL. Some of the restrictions are minor (for example, six-character external names), but others are sweeping (for example, lack of pointer variables). The following sections describe the major limitations imposed by Fortran.

External names

In Fortran, external names (procedure and common block names) cannot be longer than six characters. Furthermore, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

Procedure interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures, either by reference or by copy-in/copy-out. This flexibility of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran. That is, a procedure name can only be passed as an argument or invoked; it cannot be stored.

Pointers

The most difficult problem with Fortran is its lack of a pointer-like data type. Compiler implementation would have been far easier, and the language itself simplified considerably, if certain cases were handled by pointers. Although there are several ways of simulating pointers by using subscripts, this raises problems of external variables and initialization.

Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. As in the case of pointers, recursion can be simulated in EFL, but not without considerable effort.

Storage allocation

The definition of Fortran does not specify the lifetime of variables. It is possible but cumbersome to implement stack or heap storage disciplines by using common blocks.

14 `as` Reference

Programmers familiar with the MC68000 family of microprocessors should be able to debug code produced by `as`, the A/UX resident assembler, after reviewing this chapter—but this is not a reference for the processor itself. Details about the effects of instructions, meaning of status register bits, handling of interrupts, and many other issues are not dealt with here. This chapter should, therefore, be used in conjunction with the reference manuals in the following list.

- *MC68020 32-Bit Microprocessor User's Manual* (Prentice-Hall, 1984)
- *MC68030 Enhanced 32-Bit Microprocessor User's Manual, Second Edition* (Prentice-Hall, 1989)
- *MC68040 32-Bit Microprocessor User's Manual* (Prentice-Hall, 1991)
- *MC68851 Paged Memory Management Unit User's Manual* (Motorola, Inc., 1985)
- *MC68881/68882 Floating Point Coprocessor User's Manual* (Motorola, Inc., 1985)

Warnings

The `as` user should be aware that although, for the most part, there is a direct correspondence between `as` notation and the notation used in the documents listed in the introduction to this chapter, several exceptions could result in incorrect code. In addition to the exceptions described in the following paragraphs, refer also to the sections "Address Mode Syntax" and "Machine Instructions" later in this chapter for further information to help you avoid this.

Comparison instructions

The order of the operands in compare instructions follows one convention in the MC68000-family reference manuals and the opposite convention in `as`. Using the convention of the MC68000-family reference manuals, you might write

```
CMP.W         D5, D3       # Is D3 less than D5?
BLE           IS_LESS      # Branch if less.
```

Using the `as` convention, you would write

```
cmp.w         %d3,%d5      # Is d3 less than d5 ?
ble           is_less      # Branch if less.
```

The convention used by `as` makes for straightforward reading of compare and branch instruction sequences, with this exception: if a compare instruction is replaced by a subtract instruction, the effect on the condition codes is entirely different. This result can be confusing to programmers who are used to thinking of a comparison as a subtraction whose result is not stored. Users of `as` who become accustomed to its convention find that both the compare and subtract notations make sense in their respective contexts.

Case sensitivity

In the A/UX implementation, only lowercase instruction and register names are valid. For example,

```
mov  %d1,%d2
```

is acceptable, while

```
MOV  %D1,%D2
```

is not. This requirement is especially important for those who wish to port existing assembly code from other machines.

Overloading of opcodes

Another issue that users must be aware of arises from the MC68000-family microprocessors using several different instructions to do essentially the same thing. For example, the *MC68020 Programmer's Reference Manual* lists the instructions `SUB`, `SUBA`, `SUBI`, and `SUBQ`, which all have the effect of subtracting their source operand from their destination operand. `as` replaces the separate `suba`, `subi`, and `subq` instructions, allowing all these operations to be specified by a single assembly instruction, `sub`. On the basis of the operands given to the `sub` instruction, the `as` assembler selects the appropriate MC68000-family operation code.

The danger created by this convenience is that it can give the impression that all forms of the `SUB` operation are semantically identical when, in fact, they are not. The *MC68020 Programmer's Reference Manual* shows that while `SUB`, `SUBI`, and `SUBQ` all affect the condition codes in a consistent way, `SUBA` does not affect the condition codes at all. Consequently, the `as` user must be aware that when the destination of a `sub` instruction is an address register (which causes the `sub` to be mapped into the operation code for `suba`), the condition codes are not affected.

Using `as`

The A/UX command `as` invokes the assembler. It has the following syntax:

`as [-m] [-n] [-o` *outfile*`] [-R] [-V] [-A` *factor*`]` *filename*

The command options listed in Table 14-1 can be specified in any order.

Table 14-1 Options to `as`

Option	**Description**
`-A` *factor*	Expand the default symbol table by the factor given.
`-R`	Remove (unlink) the input file after assembly is completed; this command option is off, by default.
`-V`	Write the version number of the assembler being run on the standard error output.
`-m`*	Run the `m4` macro preprocessor on the input to the assembler.
`-n`	Turn off long/short address optimization; by default, address optimization takes place.
`-o` *outfile*	Put the output of assembly in *outfile;* by default, the output filename is formed by removing the `.s` suffix, if there is one, from the input filename and appending an `.o` suffix.
`-68030`	Assemble for the MC68030 processor and MC68030 MMU; this option gives you access to an enhanced feature set as compared with the default MC68020 assembly, but the code does not run on the original Macintosh II model computer.
`-68040`	Assemble for the MC68040 processor and MC68040 MMU; this option gives you access to an enhanced feature set as compared with the default MC68020 assembly, but the code does not run on the original Macintosh II model computer.
`-68851`	Assemble for the MC68851 Memory Management Unit (MMU); this command option is on, by default.

* If the `-m` command option is used, keywords for `m4` cannot be used as identifiers (variables, functions, labels, and so on) in the input file because `m4` cannot determine which are assembler symbols and which are real `m4` macros.

General syntax rules

The following sections discuss the components of the assembly language produced by the `as` assembler.

Format of assembly-language code

Typical lines of `as` assembly code look like these:

```
# Clear a block of memory at location %a3
   text     2
   mov.w    &const,%d1
loop:       clr.l    (%a3)+
   dbf      %d1,loop        # go back for const
                            # repetitions
init2:
   clr.l count;  clr.l credit;  clr.l debit;
```

where the suffix to `clr` is always the letter `l` (ell), while `%d1` indicates data register 1 (one).

These general points about the example should be noted:

- An identifier occurring at the beginning of a line and followed by a colon (`:`) is a **label.** In the example above, `loop` and `init2` are labels. One or more labels can precede any assembly-language instruction or pseudo-operation. Refer to the section "Location Counters and Labels" later in this chapter for additional information.
- A line of assembly code need not include an instruction. It can consist of a comment alone (introduced by `#`), or a label alone (terminated by `:`), or it can be entirely blank.
- It is good practice to use tabs to align assembly-language operations and their operands into columns, but this is not a requirement of the assembler. An opcode can appear at the beginning of the line, if desired, and spaces can precede a label. A single blank or tab suffices to separate an opcode from its operands. Additional blanks and tabs are ignored by the assembler.

- It is permissible to write several instructions on one line, separating them by semicolons. The semicolon is syntactically equivalent to a newline character; however, a semicolon inside a comment is ignored.

Comments

Comments are introduced by the character `#` and continue to the end of the line. Comments can appear anywhere; the assembler disregards them.

Identifiers

An identifier is a name for a variable, label, register, or function. It consists of a string of characters taken from the set `a-z`, `A-Z`, `_`, `~`, `%`, and `0-9`. The first character of an identifier must be a letter (uppercase or lowercase) or an underscore. The assembler distinguishes between uppercase and lowercase letters; for example, `con35`, `Con35`, and `CON35` are three distinct identifiers.

Identifiers can be up 1024 characters long (this limit is imposed by the loader).

The value of an identifier is established by the `set` pseudo-operation (refer to the section "Symbol-Definition Operations" later in this chapter) or by using it as a label (refer to the section "Location Counters and Labels" later in this chapter).

The tilde character (~) has special significance to the assembler. A tilde used alone as an identifier means "the current location." A tilde used as the first character in an identifier becomes a period (`.`) in the symbol table. This is provided for backward compatibility. The assembler now directly supports symbols such as `.eos` and `.0fake` to be entered into the symbol table, as required by the Common Object File Format. Information about file formats is provided in Section 4 of the *A/UX Programmer's Reference.*

Register identifiers

A register identifier is an identifier preceded by the character `%`. It represents one of the MC68000-family microprocessor registers. The predefined MC68020 register identifiers are shown in Table 14-2.

Table 14-2 Predefined MC68020 registers

Name	Description
`%d0-7`	Data registers
`%a0-5`	Address registers
`%a6`	Address register (also defined as `%fp`)
`%a7, %usp`	User stack pointer (also defined as `%sp`)
`%pc`	Program counter
`%ccr`	Condition code register
`%isp`	Interrupt stack pointer
`%msp`	Master stack pointer
`%sr`	Status register
`%vbr`	Vector base register
`%sfc`	Alternate function code register (also defined as `%sfcr`)
`%dfc`	Alternate function code register (also defined as `%dfcr`)
`%cacr`	Cache control register
`%caar`	Cache address register

To preserve the upward compatibility of MC68000 code, the identifiers `%a7`, `%sp`, and `%usp` represent the same machine register. Likewise, `%a6` and `%fp` are equivalent. Use of both `%a7` and `%sp`, or `%a6` and `%fp`, in the same program can result in confusion and should be avoided. The registers `%sfc` and `%sfcr` are also equivalent, as are `%dfc` and `%dfcr`.

The entire register set of the MC68000 and MC68010 is included in the MC68020 register set. Table 14-3 shows the new control registers for the MC68030. Table 14-4 shows the new control registers for the MC68040.

Various registers can be suppressed; these suppressed registers (also called *zero registers)* are used in various complex MC68020 addressing modes. The notation for suppressed registers is `%zd`*n* for data register *n*, `%za`*n* for address register *n*, and `%zpc` for the suppressed program counter.

Table 14-3 Additional registers for the MC68030 microprocessor

Name	Description
`%crp`	CPU root pointer
`%srp`	Supervisor root pointer
`%tc`	Translation control register
`%tt0, %tt1`	Translation registers
`%psr`	MMU status register

Table 14-4 Additional registers for the MC68040 microprocessor

Name	Description
`%itt0`	Instruction transparent translation register 0
`%itt1`	Instruction transparent translation register 1
`%dtt0`	Data transparent translation register 0
`%dtt1`	Data transparent translation register 1
`%mmusr`	MMU status register
`%urp`	User root pointer register

Constants

`as` deals only with integer constants. They can be entered in decimal, octal, or hexadecimal, or they can be entered as character constants. Internally, `as` treats all constants as 32-bit binary 2's-complement quantities.

Numeric constants

A decimal constant is a string of digits beginning with a nonzero digit. An octal constant is a string of digits beginning with zero. A hexadecimal constant consists of the characters `0x` or `0X` followed by a string of characters from the set `0-9`, `a-f`, and `A-F`. In hexadecimal constants, uppercase and lowercase letters are not distinguished. The following are some examples.

```
set           const,35     # decimal 35
mov.w         &035,%d1     # octal 35 (decimal 29)
set           const, 0x35  # hex 35 (decimal 53)
mov.w         &0xff,%d1    # hex ff (decimal 255)
```

Character constants

An ordinary character constant consists of a single quotation mark (`'`) followed by an arbitrary ASCII character other than the backslash (`\`). The value of the constant is equal to the ASCII code for the character. For example, the character constant `'A` has value `0x41`. Special meanings of characters are overridden when used in character constants; for example, if `'#` is used, the `#` is not treated as introducing a comment.

Special character constants convey special information to the assembler. A special character constant consists of `'\` followed by another character. All the special character constants are listed in Table 14-5.

Table 14-5 Special character constants

Constant	Value	Meaning
`'\b`	0x08	Backspace
`'\t`	0x09	Horizontal tab
`'\n`	0x0a	Newline (line feed)
`'\v`	0x0b	Vertical tab
`'\f`	0x0c	Form feed
`'\r`	0x0d	Carriage return
`'\\`	0x5c	Backslash

Other syntactic details

A discussion of expression syntax appears in the section "Expressions," later in this chapter. Information about the syntax of specific components of `as` instructions and pseudo-operations is given in the sections "Pseudo-operations," "Span-Dependent Optimization," and "Address-Mode Syntax," all later in this chapter.

Segments, location counters, and labels

The following sections describe how the assembler arranges and locates various pieces of code.

Segments

A program in `as` assembly language can be broken into segments known as `text`, `data`, and `bss` segments. The convention regarding the use of these segments is to place instructions in `text` segments, initialized data in `data` segments, and uninitialized data in `bss` segments. The assembler does not enforce this convention, however. For example, it permits intermixing of instructions and data in a `text` segment if specifically directed to mix the segments. Routines to be placed in the shared library also can have an `init` segment, which contains initialization fragments. An `init` segment is treated similarly to a `text` segment.

This convention of using separate `text`, `data`, and `bss` segments permits the sharing of `text` segments between programs on systems using shared memory, such as A/UX. If several copies of a program are running at once, which can happen when users are logged in over a network, there is only one instance of the text segment in memory, thus conserving memory space.

Primarily to simplify compiler code generation, the assembler permits up to four separate `text` segments and four separate `data` segments named `0`, `1`, `2`, and `3`. The assembly-language program can switch freely among them by using assembler pseudo-operations. (See the section "Location Counter Control Operations" later in this chapter.) This flexibility can be handy, for example, if you want to put all the constants in one segment and all the functions in another. When generating the object file, the assembler concatenates the `text` segments to generate a single `text` segment and the `data` segments to generate a single `data` segment. Thus, the object file contains only one `text` segment and only one `data` segment. There is always only one `bss` segment, and it maps directly into the object file.

Because the assembler keeps together everything from a given segment when generating the object file, the order in which information appears in the object file is unlikely to be the same as in the assembly-language file. For example, if the data for a program consists of

```
data         1            # segment 1
short        0x1111
data         0            # segment 0
long         0xffffffff
data         1            # segment 1
byte         0xff
```

the assembler groups the data for segment 0 together (in the order the assembler encounters it), then groups the data for segment 1. It then places the data for segment 1 after the data for segment 0 as it builds the object file. Thus, for the example just given, equivalent object code is generated by

```
data         1
data         0
long         0xffffffff
short        0x1111
byte         0xff
```

In this equivalent code example, the first statement

```
data         1
```

is effectively ignored.

Location counters and labels

The assembler maintains separate location counters for the `bss` segment and for each of the `text` and `data` segments. The **location counter** for a given segment is incremented by 1 for each byte generated in that segment.

The location counters allow values to be assigned to labels. When an identifier is used as a label in the assembly-language input, the value of the current location counter is assigned to the identifier. The assembler also keeps track of the segment in which the label appeared. Thus, the identifier represents a memory location relative to the beginning of a particular segment. Any label relative to the location counter should be within the text segment.

Types

Identifiers and expressions can have values of different types.

In the simplest case, an expression or identifier can have an **absolute value,** such as 29, -5000, or 262143.

◆ **Note** The term *absolute value* is not used here in the mathematical sense. ◆

An expression or identifier can have a value relative to the start of a particular segment. Such a value is known as a relocatable value. The memory location represented by such an expression cannot be known at assembly time, but the relative values of two such expressions (that is, the difference between them) can be known if they refer to the same segment.

Identifiers that appear as labels have relocatable values.

If an identifier is never assigned a value, it is assumed to be an undefined external. Such identifiers can be used with the expectation that their values are defined in another program and, therefore, known at load time, but the relative values of undefined externals cannot be known.

Expressions

Since the value of some expressions cannot be known at assembly time, expressions involving such values also cannot be known at assembly time. This section provides the rules for determining whether expressions involving unknown values can be solved.

For conciseness, the following abbreviations are useful:

abs	absolute expression
rel	relocatable expression
ext	undefined external

All constants are absolute expressions. An identifier can be thought of as an expression having the identifier type. Expressions can be built up from lesser expressions using the operators `+`, `-`, `*`, and `/`, according to the following type rules:

abs	+	*abs*	=	*abs*					
abs	+	*rel*	=	*rel*	+	*abs*	=	*rel*	
abs	+	*ext*	=	*ext*	+	*abs*	=	*ext*	
abs	-	*abs*	=	*abs*					
rel	-	*abs*	=	*rel*					
ext	-	*abs*	=	*ext*					
rel	-	*rel*	=	*abs*	(provided that the two relocatable expressions are relative to the same segment)				
abs	*	*abs*	=	*abs*					
abs	/	*abs*	=	*abs*					
-*abs*	=	*abs*							

rel - *rel* expressions are permitted only within the context of a switch statement. (See the section "Switch Table Operation" later in this chapter.) Use of a *rel* - *rel* expression is dangerous, particularly when dealing with identifiers from `text` segments. The problem is that the assembler determines the value of the expression before it resolves all questions concerning span-dependent optimizations.

The unary minus operator takes the highest precedence; the next highest precedence is given to `*` and `/`; and lowest precedence is given to `+` and binary `-`. Parentheses can be used to coerce the order of evaluation.

If the result of a division is a positive noninteger, it is truncated toward zero. If the result is a negative noninteger, the direction of truncation cannot be guaranteed.

Pseudo-operations

This section details instructions to the assembler that do not involve expressions or operators. Collectively these are known as pseudo-operations. Any pseudo-operation can be preceded by a period (`.`); this option is provided for backward compatibility.

Data initialization operations

The following pseudo-operations allocate memory space for a program:

`byte` *abs*, *abs*, ...
One or more arguments, separated by commas, can be given. The values of the arguments are computed to produce successive bytes in the assembly output.

`short` *abs*, *abs*, ...
One or more arguments, separated by commas, can be given. The values of the arguments are computed to produce successive 16-bit words in the assembly output.

`long` *expr*, *expr*, ...
One or more expressions, separated by commas, can be given. Each expression can be absolute, relocatable, or undefined external. A 32-bit quantity is generated for each such argument (in the case of relocatable or undefined external expressions, the actual value cannot be filled in until load time). Alternatively, the arguments can be bitfield expressions. A bitfield expression has the form

n:value

where both *n* and *value* denote absolute expressions. The quantity *n* represents a field width; the low-order *n* bits of *value* become the contents of the bitfield. Successive bitfields fill up 32-bit `long` quantities, starting with the high-order part. If the sum of the lengths of the bitfields is less than 32 bits, the assembler creates a 32-bit `long` with 0's filling out the low-order bits. For example,

```
long   4: -1, 16: 0x7f,  12:0, 5000
```

and

```
long   4: -1, 16: 0x7f, 5000
```

are equivalent to

```
long   0xf007f000, 5000
```

as shown in Figure 14-1.

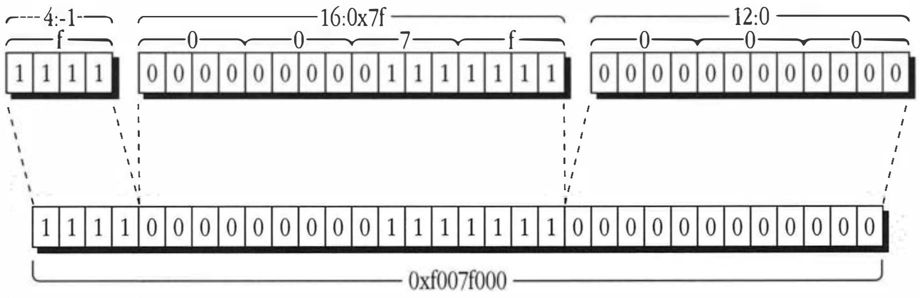

Figure 14-1 Bitfield concatenation

Bitfields cannot span pairs of 32-bit `longs`. Thus,

```
long   24: 0xa, 24: 0xb, 24:0xc
```

yields the same result as

```
long   0x00000a00, 0x00000b00, 0x00000c00
```

as shown in Figure 14-2.

`space` *abs*
The assembler computes the value of *abs* and generates the resultant number of bytes of 0 data. For example,

```
space   6
```

is equivalent to

```
byte   0,0,0,0,0,0
```

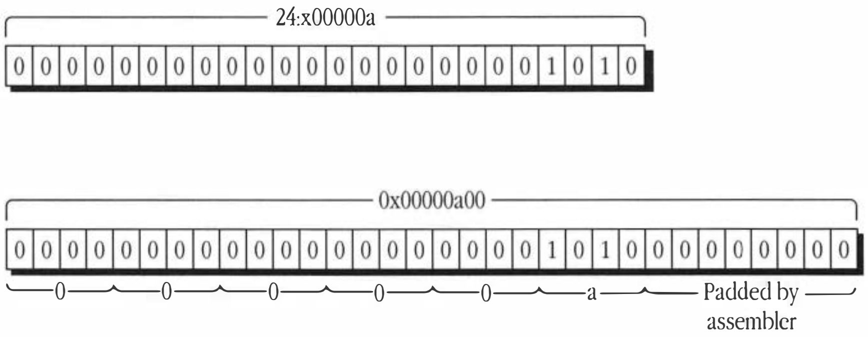

Figure 14-2 Integer boundaries

Additional pseudo-operations for MC68040 processors

You can use the following pseudo-operation to set the type of processor (CPU):

`cpu` *type*

The argument *type* can have one of the following values:

Value	*Description*
`0` or `68000`	sets the MC68000 CPU and does not set an MMU
`20` or `68020`	sets the MC68020 CPU and does not change the MMU
`30` or `68030`	sets the MC68030 CPU and MC68030 MMU
`40` or `68040`	sets the MC68040 CPU and MC68040 MMU

You also can set the type of MMU with the pseudo operation

`mmu` *type*

The argument *type* can have one of the following values:

Value	*Description*
`0`	sets no MMU
`851` or `68851`	sets the MC68851 MMU
`30` or `68030`	sets the MC68030 MMU
`40` or `68040`	sets the MC68040 MMU

Symbol-definition operations

The following pseudo-operations allocate memory space for a program:

`set` *identifier, expr*
The value of *identifier* is set equal to *expr*, which can be absolute or relocatable.

`comm` *identifier, abs*
The named *identifier* is assigned to a common area of size *abs* bytes. If *identifier* is not defined by another program, the loader allocates space for it.

`lcomm` *identifier, abs*
The named *identifier* is assigned to a local common area of size *abs* bytes. This results in allocating space in the `bss` segment. The type of *identifier* becomes relocatable.

`global` *identifier*
This causes *identifier* to be externally visible. If *identifier* is defined in the current program, declaring it `global` allows the loader to resolve references to *identifier* in other programs. If *identifier* is not defined in the current program, the assembler expects an external resolution.

Location counter control operations

The following pseudo-operations define the segment in which code is to be allocated and the alignment within that segment:

`data` *abs*
The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the `data` segment into which assembly is to be directed. If no argument is present, assembly is directed into `data` segment 0.

`text` *abs*
The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the `text` segment into which assembly is to be directed. If no argument is present, assembly is directed into `text` segment 0. Before the first `text` or `data` operation is encountered, assembly is, by default, directed into `text` segment 0.

`org` *expr*
The current location counter is set to *expr*, which must represent a value in the current segment and must not be less than the current location counter.

`even`
The current location counter is rounded up to the next even value.

`longeven`
The current location counter is rounded up to the next 4-byte multiple value.

`align` *n*
The current location counter is rounded to a multiple of *n* bytes, where *n* can be 2, 4, 8, or 16. `even` is equivalent to `align 2`; `longeven` is equivalent to `align 4`.

`init`
The assembly is directed into the `init` segment. This operation is typically used for shared-library initialization fragments.

Symbolic-debugging operations

The assembler allows for symbolic-debugging information to be placed into the object code file with special pseudo-operations. The information typically includes line numbers and information about C language symbols, such as their type and storage class. The C compilers, `cc` and `c89`, generate symbolic-debugging information when the `-g` command option is used. Assembler programmers also can include such information in source files.

`file` **and** `ln`

The `file` pseudo-operation passes the name of the source file into the object file symbol table. It has the form

`file` "*filename*"

where *filename* must be enclosed in straight double quotation marks.

The `ln` pseudo-operation makes a line-number table entry in the object file; that is, it associates a line number with a memory location. Usually the memory location is the current location in text. The format is

`ln` *line* `[,`*value*`]`

where *line* is the line number. The optional value is the address in a `text`, `data`, or `bss` segment to associate with the line number. The default when *value* is omitted (which is usually the case) is the current location in `text`.

Symbol-attribute operations

The basic symbolic testing pseudo-operations are `def` and `endef`. These operations enclose other pseudo-operations that assign attributes to a symbol and must be paired. The basic syntax for using `def` and `endef` is

```
def     name
   attrasgn
   attrasgn
     .
     .
     .
endef
```

where *attrasgn* can be any one of the attribute-assigning operations shown in the list at the end of this section.

The term `def` creates a symbol table entry but does not define the symbol. Because an undefined symbol is treated as external, a symbol that appears in a `def` pseudo-operation but never acquires a value causes an error at load time.

To allow the assembler to calculate the sizes of functions for other tools, each `def/endef` pair that defines a function name must be matched by a `def/endef` pair after the function in which a storage class of `-1` is assigned, where `-1` is the physical end of a function.

The following paragraphs describe the attribute-assigning operations (*attrasgn* in the syntax diagram just discussed). These operations apply to the symbol *name* that appeared in the opening `def` pseudo-operation.

`val` *expr*
The `val` operation assigns the value *expr* to name. The type of the expression *expr* determines with which section *name* is associated. If value is ~, the current location in the `text` section is used.

`scl` *expr*
The `scl` operation declares a storage class for *name*. The expression *expr* must yield an absolute value that corresponds to the C compiler internal representation of a storage class. The special value `-1` designates the physical end of a function.

`type` *expr*
The `type` operation declares the C language type of *name*. The expression *expr* must yield an absolute value that corresponds to the C compiler internal representation of a basic or derived type.

`tag` *expr*
The `tag` operation associates *name* with the structure, enumeration, or union named *str* that must have already been declared with a `def/endef` pair.

`line` *expr*
The `line` operation provides the line number of *name,* where *name* is a block symbol. The expression *expr* should yield an absolute value that represents a line number.

`size` *expr*
The `size` operation gives a size for *name*. The expression *expr* must yield an absolute value. When *name* is a structure or an array with a predetermined extent, *expr* gives the size in bytes. For bitfields, the size is in bits.

`dim` *expr1,expr2,* ...
The `dim` operation indicates that *name* is an array. Each of the expressions must yield an absolute value that provides the corresponding array dimension.

Switch table operation

The C compiler generates a compact set of instructions for the C language `switch` construct. For example,

```
            sub.l       &1,%d0
            cmp.l       %d0,&4
            bhi         L%21
            add.w       %d0,%d0
            mov.w       10(%pc,%d0.w),%d0
            jmp         6(%pc,%d0.w)
            swbeg       &5
L%22:
            short       L%15-L%22
            short       L%21-L%22
            short       L%16-L%22
            short       L%21-L%22
            short       L%17-L%22
```

The special `swbeg` pseudo-operation communicates to the assembler that the lines following it contain *rel* - *rel* subtractions. Ordinarily, such subtractions are risky because of span-dependent optimization. In this case, however, the assembler makes special allowances for the subtraction, because the compiler guarantees that both symbols are defined in the current assembler file and that one of the symbols is a fixed distance away from the current location.

The `swbeg` pseudo-operation takes an argument that looks like an immediate operand. The argument is the number of lines following the `swbeg` instruction that contain switch table entries. `swbeg` inserts two words into text. The first is the `illegal` instruction code. The second is the number of table entries that follow. The disassembler `dis`(1) needs the `illegal` instruction as a hint that what follows is a switch table. Otherwise, `dis`(1) tries to decode the table entries as instructions when they are really differences between two symbols.

Span-dependent optimization

The assembler chooses the object code it generates according to the distance between an instruction and its operands. Several choices are available; distances can be expressed with 8-, 16-, or 32-bit displacements. These are called the short, long, and very long forms, respectively.

Choosing the smallest, fastest form is called **span-dependent optimization.** Span-dependent optimization occurs most obviously in the choice of object code for branches and jumps. It also occurs when an operand can be represented by the program counter relative address mode instead of as an absolute two-word (`long`) address. The span-dependent optimization capability is normally enabled; the `-n` command option disables it. When this capability is disabled, the assembler makes worst-case assumptions about the types of object code that must be generated. Span-dependent optimizations are performed only within `text` segment 0. Any reference outside `text` segment 0 is assumed to be a worst case.

The C compilers, `cc` and `c89`, generate branch instructions without a specific offset size. When the optimizer is used, it identifies branches that can be represented by the short form, and it changes the operation accordingly. The assembler chooses only between long and very long representations for branches.

Although the largest offset specification allowed is a word, large programs conceivably can have need for a branch to a location that cannot be reached by a word displacement. Therefore, equivalent long forms of these instructions might be needed. When the assembler encounters a branch instruction without a size specification, it tries to choose between the long and very long forms of the instruction. If the operand can be represented in a word, the long form of the instruction is generated. Otherwise, the very long form is generated. For unconditional branches (for example, `br`, `bra`, and `bsr`), the very long form is just the equivalent jump (`jmp` and `jsr`) with an absolute (instead of `pc`-relative) address operand. For conditional branches, the equivalent very long form is a conditional branch around a jump, where the conditional test is reversed.

Table 14-6 summarizes span-dependent optimizations. Again, the assembler chooses only between the long form and the very long form, while the optimizer chooses between the short and long forms for branches (but not `bsr`).

Table 14-6 Assembler span-dependent optimizations

Instruction	Short form	Long form	Very long form
`br`, `bra`, `bsr`	Byte offset	Word offset	`jmp` or `jsr` with absolute long address
Conditional branch	Byte offset	Word offset	Short conditional branch with reversed condition around `jmp` with absolute long address
`jmp`, `jsr`		`pc`-relative address	Absolute long address
`lea`, `pea`		`pc`-relative address	Absolute long address

Branch instructions can have either a byte, word, or long `pc`-relative address operand. The assembler still chooses between word and long representations for branches if no byte size specification is given; however, the long form is replaced by a branch long with `pc`-relative address instead of a jump with absolute long address.

Address modes

The `as` assembler provides you with nine basic kinds of addressing modes:

- register direct
- register indirect
- register indirect with index
- memory indirect
- program counter indirect with displacement
- program counter indirect with index
- program counter memory indirect
- absolute
- immediate

Address-mode syntax

In the tables in the remainder of this chapter, the following abbreviations are used:

`A`*n*`/a`*n*	Address register, where *n* is any digit from 0 through 7.
bd	2's-complement base displacement that is added before indirection takes place; the size can be 16 or 32 bits.
d	2's-complement or sign-extended displacement that is added as part of effective address calculation; size can be 8 or 16 bits. (When omitted, the assembler uses the 0 value.)
`D`*n*`/d`*n*	Data register, where *n* is any digit from 0 through 7.
od	Outer displacement that is added as part of effective address calculation after memory indirection; the size can be 16 or 32 bits.
`PC/pc`	Program counter.
Ri/ *ri*	Index register *i* can be any address or data register with an optional size designation (that is, *ri*`.w` for 16 bits or *ri*`.l` for 32 bits); the default size is 16 bits (`.w`).
scl	Optional scale factor that can be multiplied times index register in some modes. (Values for *scl* are 1, 2, 4, or 8; the default is 1.)
`[ ]`	Grouping characters used to enclose an indirect expression; these are required characters. (Addressing arguments can occur in any order within the brackets.)
`( )`	Grouping characters used to enclose an entire effective address; these are required characters. (Addressing arguments can occur in any order within the parentheses.)
`{ }`	Indicate that a scale factor is optional; these are not required characters.

It is important to note that expressions used for the absolute addressing modes need not be absolute expressions in the sense previously described in the section "Types." Although the addresses used in those addressing modes must ultimately be filled in with constants, that can be done later by the loader. The assembler need not be able to compute them. Indeed, the absolute long addressing mode is commonly used for accessing undefined external addresses. Several examples follow that illustrate the use of this notation.

- `%d7` indicates data register 7.
- `(%a3)` indicates the contents of address register 3.
- `14(%a4)` indicates that the decimal displacement 14 is added to the contents of address register 4.
- `(%a1.w)` indicates the contents of the word in address register 1; the register is treated as an index register.
- `(%d3.w{*2})` indicates the contents of the word in index register `d3`; the register is treated as an index register and the contents of the register are multiplied by 2.

Effective address modes

Table 14-7 summarizes the `as` syntax for MC68000-family addressing modes. In Table 14-7, the index register notation should be understood as *ri. size* scale,* where both *size* and *scale* are optional. The MC68000-family user's manuals provide information about generating effective addresses and assembler syntax.

Note that suppressed address register `%za`*n* can be used in place of `%a`*n,* suppressed PC register `%zpc` can be used in place of `%pc`, and suppressed data register `%zd`*n* can be used in place of `%d`*n,* if you want them suppressed.

Address modes for the MC68020 use two different formats of extension. The brief format provides fast indexed addressing, while the full format provides a number of options in size of displacement and indirection. The assembler generates the brief format if the effective address expression is not memory indirect, the value of displacement is within a byte, and no base or index suppression is specified; otherwise, the assembler generates the full format.

Some variations of the MC68020 addressing modes might be redundant with the MC68000 address register indirect, address register indirect with displacement, and program counter with displacement modes. The assembler selects the more efficient mode when redundancy occurs. For example, when the assembler sees the form `(A`*n*`)`, it generates address register indirect mode.

The assembler generates address register indirect with displacement when it encounters any of the following forms (as long as *bd* fits in 16 bits or less):

bd`(A`*n*`)`
`(`*bd*`, A`*n*`)`
`(A`*n*`, `*bd*`)`

Table 14-7 Effective address modes

MC680x0 notation	`as` notation	Address mode
D*n*	%d*n*	Data register direct
A*n*	%a*n*	Address register direct
(A*n*)	(%a*n*)	Address register indirect
(A*n*)+	(%a*n*)+	Address register indirect with postincrement*
-(A*n*)	-(%a*n*)	Address register indirect with predecrement*
d(A*n*)	*d*(%a*n*)	Address register indirect with displacement (*d* signifies a signed 16-bit absolute displacement)
(A*n*, *Ri*)	(%a*n*,%*ri*.w) (%a*n*,%*ri*.l)	Address register indirect with index
d(A*n*, *Ri*)	*d*(%a*n*,%*ri*.w) *d*(%a*n*,%*ri*.l)	Address register indirect with index plus displacement (*d* signifies a signed 8-bit absolute displacement)
(A*n*, *Ri*{**scl*})	(%a*n*,%*ri*{**scl*})	Address register direct with index
(*bd*,A*n*, *Ri*{**scl*})	(*bd*, %a*n*, %*ri*{**scl*})	Address register direct with index plus base displacement
([*bd*, A*n*, *Ri*{**scl*}], *od*)	([*bd*, %a*n*,%*ri*{**scl*}], *od*)	Memory indirect with preindexing plus base and outer displacement
([*bd*,A*n*], *Ri*{**scl*}, *od*)	([*bd*, %a*n*], %*ri*{**scl*}, *od*)	Memory indirect with postindexing plus base and outer displacement
d(PC)	*d*(%pc)	Program counter indirect with displacement (*d* signifies 16-bit displacement)
d(PC, *Ri*)	*d*(%pc, %*rn*.l) *d*(%pc, %*rn*.w)	Program counter direct with index and displacement (*d* signifies 8-bit displacement)
(*bd*, PC, *Ri*{**scl*})	(*bd*, %pc, %*ri*{**scl*})	Program counter direct with index and base displacement
([*bd*,PC], *Ri*{**scl*}, *od*)	([*bd*, %pc], %*ri*{**scl*}, *od*)	Program counter memory indirect with postindexing plus base and outer displacement
([*bd*,PC, *Ri*{**scl*}], *od*)	([*bd*, %pc, %*ri*{**scl*}], *od*)	Program counter memory indirect with preindexing plus base and outer displacement
xxx.W	*xxx*	Absolute short address (*xxx* signifies an expression yielding a 16-bit memory address)
xxx.L	*xxx*	Absolute long address (*xxx* signifies an expression yielding a 32-bit memory address)

(continued)➡

Table 14-7 Effective address modes *(continued)*

MC680x0 notation	as notation	Address mode
D#*xxx*	&*xxx*	Immediate data (*xxx* signifies an absolute constant expression)

* If the address register is the stack pointer and the operand size is byte, the address is changed by 2 rather than 1 to keep the stack pointer aligned to a word boundary.

Machine instructions

The general forms of an MC68000-family microprocessor instruction are

inst

inst operand

inst operand, operand

where *inst* is the instruction followed by 0, 1, or 2 *operands*. An *operand* can be actual data (called an immediate operand), but often *operand* is the effective address of the data to be used in the instruction.

Table 14-8 shows how MC68000-family instructions should be written to ensure that the `as` assembler correctly understands them. The following abbreviations are used in Table 14-8.

A	The letter *A*, as in `add.`*A*, stands for one of the address operation size attribute letters `w` or `l`, representing a word or long operation, respectively.
CC	In the contexts `b`*CC*, `db`*CC*, and `s`*CC*, the letters *CC* represent any of the following condition code designations (except that `f` and `t` cannot be used in the `b`*CC* instruction):

`cc`	carry clear
`cs`	carry set
`eq`	equal
`f`	false
`ge`	greater or equal

	`gt`	greater than
	`hi`	high
	`hs`	high or same (`=cc`)
	`le`	less or equal
	`lo`	low (`=cs`)
	`ls`	low or same
	`lt`	less than
	`mi`	minus
	`ne`	not equal
	`pl`	plus
	`t`	true
	`vc`	overflow clear
	`vs`	overflow set
d	2's-complement or sign-extended displacement that is added as part of effective address calculation; the size can be 8 or 16 bits. (When omitted, the assembler uses value of 0.)	
EA	An arbitrary effective address.	
(*eq*)	The two forms of machine instruction are equivalent.	
FC	A function code that can be a data register, `%sfc`, `%dfc`, or an absolute expression with value 0 through 7 for MC68030 addressing or 0 through 15 for 68851 addressing.	
I	An absolute expression representing a level, 0 through 7.	
I	An absolute expression, used as an immediate operand.	
L	A label reference, or any expression representing a memory address in the current segment.	
mask	An absolute expression with value 0 through 7 for MC68030 addressing or 0 through 15 for 68851 addressing.	
offset	Either an immediate operand or a data register.	
PCC	One of the MC68851 PMMU condition codes.	
Q	An absolute expression evaluating to a number from 1 through 8.	

S	The letter *S*, as in `add.`*S*, stands for one of the operation size attribute letters `b`, `w`, or `l`, representing a byte, word, or long operation, respectively.
width	Either an immediate operand or a data register.

Registers are designated using the following components:

`%`	Register call.
`a`	Address register.
`d`	Data register.
r	Either data or address register.
x, *y*, *m*, *n*	Any digit from 0 through 7, where $x \neq y$, $m \neq n$, and $x \neq m$, and $y \neq n$.

These components are combined to form the following register designations:

`%a`*x*, `%a`*y*, `%a`*n*	Address registers.
`%d`*x*, `%d`*y*, `%d`*n*	Data registers.
`%`*mr*	(P)MMU register (`%crp`, `%srp`, `%tt0`, `%tt1`, `%drp`, `%pcsr`, `%psr`, `%cal`, `%val`, `%scc`, `%ac`, `%bad`*x*, `%bac`*x*).
`%`*rc*	Control register (`%sfc`, `%dfc`, `%cacr`, `%vbr`, `%caar`, `%msp`, `%isp`).
`%`*rx*, `%`*ry*, `%`*rn*	Either data or address registers.
(*eq*)	The two forms of machine instruction are equivalent.

Table 14-8 MC68000-family instruction formats

Mnemonic	**Assembler syntax**		**Operation**
ABCD	`abcd.b`	`%d`*y*`,%d`*x*	Add decimal with extend
	`abcd.b`	`-(%a`*y*`),-(%a`*x*`)`	
ADD	`add.`*S*	*EA*`,%d`*n*	Add binary
	`add.`*S*	`%d`*n*`,`*EA*	
ADDA	`add.`*A*	*EA*`,%a`*n*	Add address
ADDI	`add.`*S*	`&`*I*`,`*EA*	Add immediate
ADDQ	`add.`*S*	`&`*Q*`,`*EA*	Add quick
ADDX	`addx.`*S*	`%d`*y*`,%d`*x*	Add extended
	`addx.`*S*	`-(%ay),-(%ax)`	

Table 14-8 MC68000-family instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
AND	`and.`*S* `and.`*S*	*EA*`,%d`*n* `%d`*n*`,` *EA*	AND logical
ANDI	`and.`*S*	`&`*I*`,` *EA*	AND immediate
ANDI to CCR	`and.b`	`&`*I*`, %cc`	AND immediate to condition code register
ANDI to SR	`and.w`	`&`*I*`, %sr`	AND immediate to status register
ASL	`asl.`*S* `asl.`*S* `asl.w`	`%d`*x*`, %d`*y* `&`*Q*`, %d`*y* `&1,`*EA*	Arithmetic shift (left)
ASR	`asr.`*S* `asr.`*S* `asr.w`	`%d`*x*`, %d`*y* `&`*Q*`, %d`*y* `&1,` *EA*	Arithmetic shift (right)

Bcc	`b`*CC*
	`b`*CC*`.b`
	`b`*CC*`.l`
		L
		L
		L
			Branch conditionally (16-bit displacement)
			Branch conditionally (short) (8-bit displacement)
			Branch conditionally (long) (32-bit displacement)

BCHG	`bchg` `bchg`	`%d`*n*`,` *EA* `&I,` *EA*	Test a bit and change Note: `bchg` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is used
BCLR	`bclr` `bclr`	`%d`*n*`,`*EA* `&`*I*`,`*EA*	Test a bit and clear Note: `bclr` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is used.
BFCHG	`bfchg`	*EA*`{`*offset*`:`*width*`}`	Complement bitfield
BFCLR	`bfclr`	*EA*`{`*offset*`:`*width*	Clear bitfield
BFEXTS	`bfexts`	*EA*`{`*offset*`:`*width*`}, %d`*n*	Extract bitfield (signed)
BFEXTU	`bfextu`	*EA*`{`*offset*`:`*width*`}, %d`*n*	Extract bitfield (unsigned)
BFFFO	`bfffo`	*EA*`{`*offset*`:`*width*`}, %d`*n*	Find first one in bitfield
BFINS	`bfins`	`%d`*n*`,` *EA*`{`*offset*`:`*width*`}`	Insert bitfield
BFSET	`bfset`	*EA*`{`*offset*`:`*width*`}`	Set bitfield
BFTST	`bftst`	*EA*`{`*offset*`:`*width*`}`	Test bitfield
BKPT	`bkpt`	`&`*I*	Breakpoint
BRA	`bra.`*S* `br.`*S*	*L* *L*	Branch always Same as `bra.`*S*

(continued)➨

Table 14-8 MC68000-family instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
BSET	bset	%d*n*,*EA*	Test a bit and set
	bset	&*I*, *EA*	Note: bset must be written with no suffix. If the second operand is a data register, .l is assumed; otherwise, .b is used.
BSR	bsr.*S*	*L*	Branch to subroutine
BTST	btst	%d*n*,*EA*	Test a bit and set
	btst	&*I*, *EA*	Note: btst must be written with no suffix. If the second operand is a data register, .l is assumed; otherwise, .b is used.
CALLM	callm	&*I*,*EA*	Call module
CAS	cas.*S*	%d*x*, %d*y*, *EA*	Compare and swap operands
CAS2	cas2.*S*	%d*x*:%d*y*, %d*m*:%d*n*, (%*rx*):(%*ry*)	Compare and swap dual operands
CHK	chk.*A*	*EA*, %d*n*	Check register against bounds
CHK2	chk2.*S*	*EA*, %*rn*	Check register against bounds
CLR	clr.*S*	*EA*	Clear an operand
CMP	cmp.*S*	%d*n*, *EA*	Compare[*]
CMPA	cmpa.*A*	%a*n*, *EA*	Compare address[*] [†]
CMPI	cmpi.*S*	*EA*, &*I*	Compare immediate[*] [†]
CMPM	cmpm.*S*	(%a*x*)+, (%a*y*)+	Compare memory[*] [†]
CMP2	cmp2.*S*	%*rn*, *EA*	Compare register against bounds[†]
DBcc	db*CC*	%d*n*, *L*	Test condition, decrement, and branch
	dbra	%d*n*, *L*	Decrement and branch always
DIVS	divs.w	*EA*, %d*x*	Signed divide 32/16 –> 16r: 16q
	tdivs.l	*EA*, %d*x*	Signed divide (long) 32/32 –> 32q
	divs.l	*EA*, %d*x*	
	divs.l	*EA*, %d*x*:%d*y*	Signed divide (long) 32/32 –> 32r:32q[‡]
DIVSL	tdivs.l	*EA* , %d*x*: %d*y*	Signed divide (long) 64/32 –> 32r:32q
DIVU	divu.w	*EA*, %d*n*	Unsigned divide 32/16 –> 16r:16q
	tdivu.l	*EA*, %d*x*	Unsigned divide (long) 32/32 –> 32*(eq)*
	divu.l	*EA*, %d*x*	
DIVUL	divu.l	*EA* , %d*x*: %d*y*	Unsigned divide (long) 64/32 –> 32r:32q[§]
	tdivu.l	*EA* , %d*x*: %d*y*	Unsigned divide (long) 32/32 –> 32r:32q[‖]

Table 14-8 MC68000-family instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
EOR	eor.*S*	%d*n*,*EA*	Exclusive OR logical
EORI	eor.*S*	&*I*, *EA*	Exclusive OR immediate
EORI to CCR	eor.b	&*I*,%cc	Exclusive OR immediate to condition code register
EORI to SR	eor.w	&*I*,%sr	Exclusive OR immediate to the status register
EXG	exg	%*rx*,%*ry*	Exchange registers
EXT	ext.w	%d*n*	Sign-extend low-order byte of data to word
	ext.l	%d*n*	Sign-extend low-order word of data to long
EXTB	extw.l	%d*n*	Same as ext.l
	extb.l	%d*n*	Sign-extend low-order byte of data to long
ILLEGAL	illegal		Illegal instruction
JMP	jmp	*EA*	Jump
JSR	jsr	*EA*	Jump to subroutine
LEA	lea	*EA*,%a*n*	Load effective address
LINK	link.*A*	%a*n*,&*I*	Link and allocate
LSL	lsl.*S*	%d*x*,%d*y*	Logical shift (left)
	lsl.*S*	&*Q*,%d*y*	
	lsl.*S*	*EA*	
LSR	lsr.*S*	%d*x*,%d*y*	Logical shift (right)
	lsr.*S*	&*Q*,&d*y*	
	lsr.*S*	*EA*	
MOVE	move.*S*	*EA*,*EA*	Move data from source to destination# **
MOVE16	move16	*EA*,*EA*	Move 16 byte block from source to destination# ††
MOVE to CCR	move.w	*EA*,%cc	Move to condition code register#
MOVE from CCR	move.w	%cc,*EA*	Move from condition code register#
MOVE to SR	move.w	*EA*,%sr	Move to the status register#
MOVE from SR	move.w	%sr,*EA*	Move from the status register#
MOVE USP	move.l	%usp,%a*n*	Move user stack pointer#
	move.l	%a*n*,%usp	
MOVEA	move.*A*	*EA*,%a*n*	Move address#
MOVEC	move.l	%*rc*,%*rn*	Move from/to control register#
	move.l	%*rn*,%*rc*	

(continued)➡

Table 14-8 MC68000-family instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
MOVEM	movem.*A*	*EA*,&*I*	Move multiple registers# ⌗
	movem.*A*	&*I*,*EA*	
MOVEP	movep.*A*	%d*x*,*d*(%a*y*)	Move peripheral data#
	movep.*A*	*d*(%a*y*),%d*x*	
MOVEQ	move.l	&*I*,%d*n*	Move quick#
MOVES	moves.*S*	%*rn*,*EA*	Move to/from address space#
	moves.*S*	EA,%*rn*	
MULS	muls.w	EA,%d*x*	Signed multiply 16*16 –> 32
	tmuls.l	EA,%d*x*	Signed multiply (long) 32*32 –> 32 (*eq*)
	muls.l	EA,%d*x*	
	muls.l	EA,%d*x*:%d*y*	Signed multiply (long) 32*32 –> 64
MULU	mulu.w	EA,%d*x*	Unsigned multiply 16*16 –> 32
	tmulu.l	EA,%d*x*	Unsigned multiply (long) 32*32 –> 32(*eq*)
	mulu.l	EA,%d*x*	
	mulu.l	EA,%d*x*:%d*y*	Unsigned multiply (long) 32*32 –> 64
NBCD	nbcd	*EA*	Negate decimal with extend
NEG	neg.*S*	*EA*	Negate
NEGX	negx.*S*	*EA*	Negate with extend
NOP	nop		No operation
NOT	not.*S*	*EA*	Logical complement
OR	or.*S*	EA,%d*n*	Inclusive OR logical
	or.*S*	%d*n*,*EA*	
ORI	ori.*S*	&*I*,*EA*	Inclusive OR immediate; equivalent to or.*S*
ORI to CCR	ori.w	&*I*,%cc	Inclusive OR immediate to condition code register; equivalent to or.w
ORI to SR	ori.w	&*I*,%sr	Inclusive OR immediate to the status register; equivalent to or.w
PACK	pack	-(%a*x*),-(%a*y*),&*I*	Pack BCD
	pack	%d*x*,%d*y*,&*I*	
PEA	pea	*EA*	Push effective address
RESET	reset		Reset external devices

Table 14-8 MC68000-family instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
ROL	`rol.`*S*	`%d`*x*`,%d`*y*	Rotate left without extend
	`rol.`*S*	`&`*Q*`,%d`*y*	
	`rol.w`	*EA*	
ROR	`ror.`*S*	`%d`*x*`,%d`*y*	Rotate right without extend
	`ror.`*S*	`&`*Q*`,%d`*y*	
	`ror.w`	*EA*	
ROXL	`roxl.`*S*	`%d`*x*`,%d`*y*	Rotate left with extend
	`roxl.`*S*	`&`*Q*`,%d`*y*	
	`roxl.w`	*EA*	
ROXR	`roxr.`*S*	`%d`*x*`,%d`*y*	Rotate right with extend
	`roxr.`*S*	`&`*Q*`,%d`*y*	
	`roxr.w`	*EA*	
RTD	`rtd`	`&`*I*	Return and deallocate parameters
RTE	`rte`	`%`*rn*	Return from exception
RTM	`rtm`		Return from module
RTR	`rtr`		Return and restore condition codes
RTS	`rts`		Return from subroutine
SBCD	`sbcd`	`%d`*y*`,%d`*x*	Subtract decimal with extend
	`sbcd`	`-(%a`*y*`), -(%a`*x*`)`	
Scc	`s`*CC*	*EA*	Set according to condition
STOP	`stop`	`&`*I*	Load status register and stop
SUB	`sub.`*S*	`EA,%d`*n*	Subtract binary
		`%d`*n*`,`*EA*	
SUBA	`sub.`*A*	`EA,%a`*n*	Subtract address
SUBI	`sub.`*S*	`&`*I*`,` *EA*	Subtract immediate (`subi` also works)
SUBQ	`sub.`*S*	`&`*Q*`,` *EA*	Subtract quick (`subq` also works)
SUBX	`subx.`*S*	`%d`*y*`,%d`*x*	Subtract with extend
		`-(%a`*y*`), -(%a`*x*`)`	
SWAP	`swap`	`%d`*n*	Swap register halves
TAS	`tas`	*EA*	Test and set an operand
TRAP	`trap`	`&`*I*	Trap
TRAPV	`trapv`		Trap on overflow

(continued)➡

Table 14-8 MC68000-family instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
TRAPcc	`t`*CC*		Trap on condition (*eq*)
	`trap`*CC*		
	`tp`*CC*`.`*A*	`&`*I*	
	`trap`*CC*`.`*A*	`&`*I*	(*eq*)
TST	`tst.`*S*	*EA*	Test an operand
UNLK	`unlk`	`%a`*n*	Unlink
UNPK	`unpk`	`-(%a`*x*`), -(%a`*y*`),&`*I*	Unpack BCD
	`unpk`	`%d`*x*`, %d`*y*`,&`*I*	

* The order of operands in `as` is the reverse of that in the *MC68881 Programmer's Reference Manual*.

† The `cmp.`*S* syntax is also recognized.

‡ Whenever `%d`*x* and `%d`*y* are the same register, the form is equivalent to the `divs.l` *EA*`, %d`*x* form.

§ Whenever `%d`*x* and `%d`*y* are the same register, the form is equivalent to the `divu.l` *EA*`, %d`*x* form.

∥ Whenever `%d`*x* and `%d`*y* are the same register, the form is equivalent to the `tdivu.l` *EA*`, %d`*x* form.

# In all move commands, `move` can be shortened to `mov`.

** If the destination is an address register, the instruction generated is `MOVEA`.

†† This instruction is available on the MC68040 only.

‡‡ The immediate operand is a mask designating which registers are to be moved to memory or which are to receive memory data. Not all addressing modes are permitted and the correspondence between mask bits and register numbers depends on the addressing mode.

Instructions for the MC68881

All Macintosh computers that run A/UX are equipped with floating-point coprocessor capability; this is embedded in the microprocessor on systems with a 68040 or higher microprocessor. The coprocessor uses special instructions that are introduced in this section.

A/UX and the MC68881 coprocessor fully support the IEEE standard for handling NaN (Not a Number) conditions. To maximize flexibility there are two modes of handling unordered conditions: non-IEEE and IEEE. These condition codes are shown in Tables 14-9 and 14-10, respectively. Non-IEEE condition codes are used in the two following cases.

- when porting a program that does not support the IEEE standard
- when generating code that does not support the IEEE floating-point concepts (for example, the unordered condition)

When non-IEEE condition codes are used, an exception is generated when an unordered condition is found. It is the responsibility of the application to test for and handle these conditions. Generally, A/UX users want to use the IEEE condition codes.

Table 14-9 Non-IEEE condition codes

CC	Meaning
`ge`	Greater than or equal
`gl`	Greater or less than
`gle`	Greater or less than or equal
`gt`	Greater than
`le`	Less than or equal
`lt`	Less than
`ngt`	Not greater than
`nge`	Not greater than or equal
`nlt`	Not less than
`ngl`	Not greater or less than
`nle`	Not less than or equal
`ngle`	Not greater or less than or equal
`sneq`	Signaling not equal
`sf`	Signaling false
`seq`	Signaling equal
`st`	Signaling true

Table 14-10 IEEE condition codes

CC	Meaning
`eq`	Equal
`oge`	Ordered greater than or equal
`ogl`	Ordered greater or less than
`ogt`	Ordered greater than
`ole`	Ordered less than or equal
`olt`	Ordered less than
`or`	Ordered
`t`	True
`ule`	Unordered or less or equal
`ult`	Unordered or less than
`uge`	Unordered or greater than or equal
`ueq`	Unordered or equal
`ugt`	Unordered or greater than
`un`	Unordered
`neq`	Not equal
`f`	False

The floating-point coprocessor also supports a set of standard constants that are kept in ROM. The MC68881 constant ROM values are shown in Table 14-11. In this table, *ccc* indicates a *constant condition code designator*.

Table 14-11 Constants in MC68881 ROM

ccc	Value	ccc	Value
0x0	pi	3x5	10**4
0xB	log10(2)	3x6	10**8
0xC	e	3x7	10**16
0xD	log2(e)	3x8	10**32
0xE	log10(e)	3x9	10**64
0xF	0.0	3xA	10**128
3x0	ln(2)	3xB	10**256
3x1	ln(10)	3xC	10**512
3x2	10**0	3xD	10**1024
3x3	10**1	3xE	10**2048
3x4	10**2	3xF	10**4096

Table 14-12 shows how the floating-point coprocessor (MC68881) instructions should be written to be understood by the `as` assembler. Abbreviations used in Table 14-12 are as follows:

A	source format letters `w` or `l`
B	source format letters `b`, `w`, `l`, `s`, or `p`
CC	any of the floating-point condition code designations listed in Table 14-9 and Table 14-10
ccc	any of the ROM constants listed in Table 14-11
EA	an effective address
I	an absolute expression, used as an immediate operand
L	a label reference or any expression representing a memory address in the current segment

SF	source format letters: `b` = byte integer `d` = double precision `l` = long word integer `p` = packed binary code decimal `s` = single precision `w` = word integer `x` = extended precision
R	rounding precision letters: `s` = single precision `d` = double precision

◆ **Note** The source format must be specified if more than one source format is permitted, otherwise a default source format of extended precision (`x`) is assumed. Source format need not be specified if only one format is permitted by the operation. ◆

`%`*control*	floating-point control register
`%d`*n*	data register, where $0 \leq n \leq 7$
`%fpcr`	floating-point control register
`%fpiar`	floating-point instruction address register
`%fp`*m*, `%fp`*n*, `%fp`*q*	floating-point data registers, where *m*, *n*, and *q* are digits from 0 through 7
`%fpsr`	floating-point status register
`%`*iaddr*	floating-point instruction address register
`%`*status*	floating-point status register

Table 14-12 Floating-point instruction formats

Mnemonic	Assembler syntax		Operation
FABS	`fabs.`*SF*	*EA*`,%fp`*n*	Absolute value function
	`fRabs.`*SF*	*EA*`,%fp`*n*	
	`fabs.x`	`%fp`*m*`,%fp`*n*	
	`fRabs.x`	`%fp`*m*`,%fp`*n*	
	`fabs.x`	`%fp`*n*	
	`fRabs.x`	`%fp`*n*	
FACOS	`facos.`*SF*	*EA*`,%fp`*n*	Arccosine function
	`facos.x`	`%fp`*m*`,%fp`*n*	
	`facos.x`	`%fp`*n*	
FADD	`fadd.`*SF*	*EA*`,%fp`*n*	Floating-point add
	`fadd.`*SF*	*EA*`,%fp`*n*	
	`fadd.x`	`%fp`*m*`,%fp`*n*	
	`fRadd.x`	`%fp`*m*`,%fp`*n*	
FASIN	`fasin.`*SF*	*EA*`,%fp`*n*	Arcsine function
	`fasin.x`	`%fp`*m*`,%fp`*n*	
	`fasin.x`	`%fp`*n*	
FATAN	`fatan.`*SF*	*EA*`,%fp`*n*	Arctangent function
	`fatan.x`	`%fp`*m*`,%fp`*n*	
	`fatan.x`	`%fp`*n*	
FATANH	`fatanh.`*SF*	*EA*`,%fp`*n*	Hyperbolic arctangent function
	`fatanh.x`	`%fp`*m*`,%fp`*n*	
	`fatanh.x`	`%fp`*n*	
FBcc	`fb`*CC.A*	*L*	Coprocessor branch conditionally
FCMP	`fcmp.`*SF*	`%fp`*n*`,`*EA*	Floating-point compare*
	`fcmp.x`	`%fp`*n*`,%fp`*m*	
FCOS	`fcos.`*SF*	*EA*`,%fp`*n*	Cosine function
	`fcos.x`	`%fp`*m*`,%fp`*n*	
	`fcos.x`	`%fp`*n*	
FCOSH	`fcosh.`*SF*	*EA*`,%fp`*n*	Hyperbolic cosine function
	`fcosh.x`	`%fp`*m*`,%fp`*n*	
	`fcosh.x`	`%fp`*n*	
FDBcc	`fdb`*CC*`.w`	`%d`*n*`,`*L*	Decrement and branch on condition
FDIV	`fdiv.`*SF*	*EA*`,%fp`*n*	Floating-point divide
	`fRdiv.`*SF*	*EA*`,%fp`*n*	
	`fdiv.x`	`%fp`*m*`,%fp`*n*	
	`fRdiv.x`	`%fp`*m*`,%fp`*n*	

(continued)➡

Table 14-12 Floating-point instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
FFETOX	`fetox.`*SF*	*EA*`,%fp`*n*	e^{x} function
	`fetox.x`	`%fp`*m*`,%fp`*n*	
	`fetox.x`	`%fp`*n*	
FETOXM1	`fetoxm1.`*SF*	*EA*`,%fp`*n*	$e^{x\,(x-1)}$ function
	`fetoxm1.x`	`%fp`*m*`,%fp`*n*	
	`fetoxm1.x`	`%fp`*n*	
FGETEXP	`fgetexp.`*SF*	*EA*`,%fp`*n*	Get the exponent function
	`fgetexp.x`	`%fp`*m*`,%fp`*n*	
	`fgetexp.x`	`%fp`*n*	
FGETMAN	`fgetman.`*SF*	*EA*`,%fp`*n*	Get the mantissa function
	`fgetman.x`	`%fp`*m*`,%fp`*n*	
	`fgetman.x`	`%fp`*n*	
FINT	`fint.`*SF*	*EA*`,%fp`*n*	Integer part function
	`fint.x`	`%fp`*m*`,%fp`*n*	
	`fint.x`	`%fp`*n*	
FINTRZ	`fintrz.`*SF*	*EA*`,%fp`*n*	Integer part, round-to-zero function
	`fintrz.x`	`%fp`*m*`,%fp`*n*	
	`fintrz.x`	`%fp`*n*	
FLOG2	`flog2.`*SF*	*EA*`,%fp`*n*	Binary log function
	`flog2.x`	`%fp`*m*`,%fp`*n*	
	`flog2.x`	`%fp`*n*	
FLOG10	`flog10.`*SF*	*EA*`,%fp`*n*	Common log function
	`flog10.x`	`%fp`*m*`,%fp`*n*	
	`flog10.x`	`%fp`*n*	
FLOGN	`flogn.`*SF*	*EA*`,%fp`*n*	Natural log function
	`flogn.x`	`%fp`*m*`,%fp`*n*	
	`flogn.x`	`%fp`*n*	
FLOGNP1	`flognp1.`*SF*	*EA*`,%fp`*n*	Natural log (x+1) function
	`flognp1.x`	`%fp`*m*`,%fp`*n*	
	`flognp1.x`	`%fp`*n*	
FMOD	`fmod.`*SF*	*EA*`,%fp`*n*	Floating-point modulo
	`fmod.x`	`%fp`*m*`,%fp`*n*	
FMOVE	`mov.`*SF*	*EA*`,%fp`*n*	Move to floating-point register†
	`fmov.x`	`%fp`*m*`,%fp`*n*	
	`fRmov.x`	`%fp`*m*`,%fp`*n*	

Table 14-12 Floating-point instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
	`fmove.`*SF*	`%fp`*n*, *EA*	Move from floating-point register to memory†
	`fmove.`*SF*	`%fp`*n*, *EA*	
	`fmove.p`	`%fp`*n*, *EA*`{&`*I*`}`	
	`fmove.p`	`%fp`*n*, *EA*`{%d`*n*`}`	
	`fmove.l`	*EA*, `%`*control*	Move from memory to special register†
	`fmove.l`	*EA*, `%`*status*	
	`fmove.l`	*EA*, `%`*iaddr*	
	`fmove.l`	`%`*control*, *EA*	Move to memory from special register†
	`fmove.l`	`%`*status*, *EA*	
	`fmove.l`	`%`*iaddr*, *EA*	
FMOVECR	`fmovcr.x`	`&`*ccc*, `%fp`*n*	Move a ROM-stored value to a floating-point register† ‡ §
FMOVEM	`fmovem.x`	*EA*, `&`*I*	Move to multiple floating-point register† ‡
	`fmovem.x`	`&`*I*, *EA*	Move from multiple registers to memory† ‡
	`fmovem.x`	*EA*, `%d`*n*	Move to a data register†
	`fmovem.x`	`%d`*n*, *EA*	Move a data register to memory†
	`fmovem.l`	`%`*control*, *EA*	Move to special registers (1, 2, or 3 registers, separated by commas)†
	`fmovem.l`	`%`*status*, *EA*	
	`fmovem.l`	`%`*iaddr*, *EA*	
	`fmovem.l`	*EA*, `%`*control*	Move from special registers (1, 2, or 3 registers, separated by commas)†
	`fmovem.l`	*EA*, `%`*status*	
	`fmovem.l`	*EA*, `%`*iaddr*	
FMUL	`fmul.`*SF*	*EA*, `%fp`*n*	Floating-point multiply
	`fmul.`*SF*	*EA*, `%fp`*n*	
	`fmul.x`	`%fp`*m*, `%fp`*n*	
	`fRmul.x`	`%fp`*m*, `%fp`*n*	
FNEG	`fneg.`*SF*	*EA*, `%fp`*n*	Negate function
	`fneg.`*SF*	*EA*, `%fp`*n*	
	`fneg.x`	`%fp`*m*, `%fp`*n*	
	`fRneg.x`	`%fp`*m*, `%fp`*n*	
	`fneg.x`	`%fp`*n*	
	`fRneg.x`	`%fp`*n*	
FNOP	`fnop`		Floating-point no-op
FREM	`frem.`*SF*	*EA*, `%fp`*n*	Floating-point remainder
	`frem.x`	`%fp`*m*, `%fp`*n*	
FRESTORE	`frestore`	*EA*	Restore internal state of coprocessor

(continued)➡

Table 14-12 Floating-point instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
FSAVE	fsave	*EA*	Coprocessor save
FSCALE	fscale.*SF*	*EA*,%fp*n*	Floating-point scale exponent
	fscale.x	%fp*m*,%fp*n*	
FScc	fs*CC*.b	*EA*	Set on condition
FSGLDIV	fsgldiv.*B*	*EA*,%fp*n*	Floating-point single-precision divide
	fsgldiv.s	%fp*m*,%fp*n*	
FSGLMUL	fsglmul.*B*	fsglmul.s	Floating-point single-precision multiply
	fsglmul.s	%fp*m*,%fp*n*	
FSIN	fsin.*SF*	*EA*,%fp*n*	Sine function
	fsin.x	%fp*m*,%fp*n*	
	fsin.x	%fp*n*	
FSINCOS	fsincos.*SF*	*EA*,%fp*n*:%fp*q*	Sine/cosine function
	fsincos.x	%fp*m*,%fp*n*:%fp*q*	
FSINH	fsinh.*SF*	*EA*,%fp*n*	Hyperbolic sine function
	fsinh.x	%fp*m*,%fp*n*	
	fsinh.x	%fp*n*	
FSQRT	fsqrt.*SF*	*EA*,%fp*n*	Square root function
	fRsqrt.*SF*	*EA*,%fp*n*	
	fsqrt.x	%fp*m*,%fp*n*	
	fRsqrt.x	%fp*m*,%fp*n*	
	fsqrt.x	%fp*n*	
	fRsqrt.x	%fp*n*	
FSUB	fsub.*SF*	*EA*,%fp*n*	Floating-point subtract
	fRsub.*SF*	*EA*,%fp*n*	
	fsub.x	%fp*m*,%fp*n*	
	fRsub.x	%fp*m*,%fp*n*	
FTAN	ftan.*SF*	*EA*,%fp*n*	Tangent function
	ftan.x	%fp*m*,%fp*n*	
	ftan.x	%fp*n*	
FTANH	ftanh.*SF*	*EA*,%fp*n*	Hyperbolic tangent function
	ftanh.x	%fp*m*,%fp*n*	
	ftanh.x	%fp*n*	
FTENTOX	ftentox.*SF*	*EA*,%fp*n*	10^x function
	ftentox.x	%fp*m*,%fp*n*	
	ftentox.x	%fp*n*	
FTcc	ft*CC*		Trap on condition without a parameter

Table 14-12 Floating-point instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
FTPcc	`ftp`*CC.A*	&*I*	Trap on condition with a parameter
FTRAPcc	`ftrap`*CC*		Trap on condition without a parameter
FTRAPcc	`ftrap`*CC.A*	&*I*	Trap on condition with a parameter
FTST	`ftest.`*SF*	*EA*	Floating-point test an operand **
	`ftest.x`	`%fp`*m*	
	`ftst.`*SF*	*EA*	
	`ftst.x`	`%fp`*m*	
FTWOTOX	`ftwotox.`*SF*	*EA*, `%fp`*n*	2^xfunction
	`ftwotox.x`	`%fp`*m*, `%fp`*n*	
	`ftwotox.x`	`%fp`*n*	

* The order of operands in `as` is the reverse of that in the *MC68881 Programmer's Reference Manual.*

**The `ftst` form (floating-point trap on signal true) is no longer supported due to a conflict with the FTST (floating-point test-an-operand instruction).

† In all (floating-point) move commands, `move` may be shortened to `mov`.

‡ The immediate operand is a mask designating which registers are to be moved to memory or which registers are to receive memory data. Not all addressing modes are permitted, and the correspondence between mask bits and register numbers depends on the addressing mode used.

§ See Table 14-11, "Constants in MC68881 ROM."

Instructions for the MC68000-family MMUs

The tables in this section show how instructions for the memory management unit (the paged memory management unit, PMMU, in the Macintosh II computer) should be written to be understood by the `as` assembler.

The conditions that the memory management unit tests can be either set or cleared. Tables 14-13 and 14-14 show the mnemonics for these states. In Table 14-15, *CC* represents any of the condition code designations listed in Tables 14-13 and 14-14.

Table 14-13 Memory management condition codes: Condition is set

CC	Meaning
`bs`	Bus error
`ls`	Limit violation
`ss`	Supervisor violation
`as`	Access level violation
`ws`	Write protected
`is`	Invalid
`gs`	Gate
`cs`	Globally shared

Table 14-14 Memory management condition codes: Condition is clear

CC	Meaning
`bc`	Bus error
`lc`	Limit violation
`sc`	Supervisor violation
`ac`	Access level violation
`wc`	Write protected
`ic`	Invalid
`gc`	Gate
`cc`	Globally shared

Additional abbreviations used in Table 14-15 are as follows:

D	represents an absolute expression used as an immediate operand depth level in the `PTESTR/PTESTW` instructions, where $0 \leq D \leq 7$
EA	represents an effective address

FC	represents one of the following function codes: *I* an absolute expression used as an immediate operand `%dfc` the destination function code register `%d`*n* a data register `%sfc` the source function code register `%sfcr` the source function code register
I	represents an absolute expression used as an immediate operand
L	a label reference or any expression representing a memory address in the current segment
M	represents an absolute expression used as an immediate operand mask in the `PFLUSH/PFLUSHS` instructions, where $0 \leq M \leq 15$
`%a`*n*	represents an address register 0 through 7
`%d`*n*	represents a data register 0 through 7
`%`*pm*	represents one of the following PMMU registers: `%ac` access control register `%bac` breakpoint acknowledge control register 0 through 7 `%bad` breakpoint acknowledge data register 0 through 7 `%cal` current access level register `%crp` CPU root pointer register `%drp` DMA root pointer register `%pcsr` cache status register `%psr` status register `%scc` stack change control register `%srp` supervisor root pointer register `%tc` transition control register `%val` validate access level register

◆ **Note** The source format must be specified if more than one source format is permitted, otherwise a default source format of `w` is assumed. The source format need not be specified if only one format is permitted by the operation. ◆

Table 14-15 MMU instruction formats

Mnemonic	Assembler syntax		Operation
PB*cc*	`pb`*CC*.*A*	*L*	Branch on PMMU condition
PDB*cc*	`pdb`*CC*`.w`	`%d`*n*`,`*L*	Test, decrement, branch
PFLUSH	`pflush`	*FC*`,&`*M*	Invalidate the set of ATC entries with the given function code
	`pflush`	*FC*`,&`*M*`,`*EA*	Invalidate the set of ATC entries with the given function code and effective address
PFLUSHA	`pflusha`		Invalidate all ATC entries
PFLUSHR	`pflushr`	*EA*	Invalidate ATC and RPT entries matching effective address*
PFLUSHS	`pflushs`	*FC*`,&`*M*	Invalidate the set of ATC entries with the given function code, even if SGS bit is set*
	`pflushs`	*FC*`,&`*M*`,`*EA*	Invalidate the set of ATC entries with the given function code and effective address, even if SGS bit is set
PLOADR	`ploadr`	*FC*, *EA*	Load an entry into the ATC for read access
PLOADW	`ploadw`	*FC*, *EA*	Load an entry into the ATC for write access
PMOVE	`pmove.`*A*	`%`*pm*`,`*EA*	Move data from PMMU register to destination†
	`pmove.`*A*	*EA*`,%`*pm*	Move data from destination to PMMU register*
PMOVEFD	`pmove`	*EA*, `%`*pm*	Move data from destination to MMU register‡
PRESTORE	`prestore`	*EA*	Restore function*
PSAVE	`psave`	*EA*	Save function*
PScc	`ps`*CC*	*EA*	Set on PMMU condition
PTESTR	`ptestr`	*FC*`,`*EA*`,&`*D*	Get information about logical address; set bit for read
	`ptestr`	*FC*`,`*EA*`,&`*D*`,%an`	Get information about logical address and load register; set bit for read
PTESTW	`ptestw`	*FC*`,`*EA*`,&`*D*	Get information about logical address; set bit for write
	`ptestw`	*FC*`,`*EA*`,&`*D*`,%an`	Get information about logical address and load register; set bit for write
PTRAPcc	`pt`*CC* `ptrap`*CC* `pt`*CC*`.`*A* `ptrap`*CC*`.`*A*		Trap on PMMU condition
PTRAPpcc	`ptrap`	*PCC*	Trap on PMMU condition*

Table 14-15 MMU instruction formats *(continued)*

Mnemonic	Assembler syntax		Operation
PVALID	`pvalid`	`%val`, *EA*	Validate a pointer against VAL register*
	`pvalid`	`%a`*n*, *EA*	Validate a pointer against address register*

* These instructions are available on the MC68851 only.

† The `pmov.` syntax is also recognized.

‡ This instruction is available on the MC68030 only.

15 `ld` Reference

This chapter describes the A/UX link editor, `ld`, which creates executable object files by combining object files, performing relocation, and resolving external references. `ld` also processes symbolic debugging information. The input to `ld` is made up of relocatable object files produced by a compiler, an assembler, or a previous `ld` run. The link editor combines these object files to form either a relocatable or an absolute (executable) object file. In other documentation, the link editor is also called a *loader*.

`ld` supports a command language that lets you control the loading process with great flexibility and precision. Although the link-edit process is controlled in detail through use of this language (described later), most programmers do not require this degree of flexibility. The manual page `ld`(1) in *A/UX Command Reference* provides detailed instruction in the use of this command. This chapter is a reference to enable you to determine what functions `ld` performs.

The command language allows the link editor to perform the following functions:

- specify the machine memory configuration
- combine object file sections in particular fashions
- cause the files to be bound to specific addresses or within specific portions of memory
- define or redefine global symbols at load time

Using `ld`

To use the link editor, give the following command:

```
ld [options] filename ...
```

Files passed to `ld` must be object files, archive libraries containing object files, or text source files containing `ld` directives. `ld` uses the *magic number* of the file (the first 2 bytes of the file) to determine the type of the file. If `ld` does not recognize the magic number, it assumes the file is a text file containing `ld` directives and attempts to parse it.

Input object files and archive libraries of object files are loaded together to form an output object file. If there are no unresolved references, the file should be executable. For additional information, see the section "Object Files" later in this chapter.

Object files have the form *name*`.o` throughout the examples in this chapter. The names of actual input object files need not follow this convention.

If you merely want to load the object files *file1*`.o` and *file2*`.o`, use the command:

```
ld file1.o file2.o
```

No directives to `ld` are needed. If no errors are encountered during the load, the output is left in the default file `a.out`.

`ld` combines the input file sections in order. That is, if each of *file1*`.o` and *file2*`.o` contains the standard sections `.text`, `.data`, and `.bss`, the output object file also contains these three sections. `ld` creates the output `.text` section by concatenating the `.text` sections from *file1*`.o` and *file2*`.o`. The `.data` and `.bss` sections are formed similarly. Then, `ld` binds the output `.text` section at address 0x000000. The output `.data` and `.bss` sections are loaded together into contiguous addresses.

Instead of entering the names of files to be loaded, or entering `ld` options on the `ld` command line, you can place this information in a separate file and simply pass the file to `ld`. Such an input file containing link editor directives is referred to as an *i-file* in this chapter. Its usefulness is explained in the paragraphs that follow `.ld` automatically searches for an i-file named `default.ld` in the list of library directories (see the `-l` and `-L` options under "Options," later in this chapter). The default directory for this search is `/usr/lib`.

For example, if you frequently load the object files *file1*`.o`, *file2*`.o`, and *file3*`.o` with the same options *f1* and *f2*, you can enter the command

`ld -`*f1* `-`*f2* *file1*`.o` *file2*`.o` *file3*`.o`

each time you must invoke `ld`. Alternatively, you could create an i-file containing the statements

-f
-f2
file1`.o`
file2`.o`
file3`.o`

and use the following command:

`ld` *i-file*

Note that it is permissible to specify some of the object files to be loaded in the i-file and to specify others on the command line, as well as to specify some options in the i-file and others on the command line. Note also that either white space or newlines can separate the statements in an i-file. Input object files are loaded in the order they are encountered, whether on the command line or in an i-file. As an example, if a command line were

`ld` *file1*`.o` *i-file* *file2*`.o`

and the i-file contained

file3`.o`
file4`.o`

the order of loading would be

1. *file1*`.o`
2. *file3*`.o`
3. *file4*`.o`
4. *file2*`.o`

Note from this example that an i-file is read and processed immediately upon being encountered in the command line.

Link editor concepts

There are several concepts and definitions that can help you become familiar with the link editor.

Memory configuration

The virtual memory of an A/UX system is, for purposes of allocation, partitioned into configured memory and unconfigured memory. *Configured memory* indicates a range of memory for which the appropriate single in-line memory modules (SIMMs) are installed and available for use. *Unconfigured memory* denotes a range of memory for which no chips are installed or that is reserved by the operating system or the NuBus address space. The default is to treat all memory as configured.

◆ **Note** Nothing can be loaded into unconfigured memory. ◆

Specifying a certain memory range as unconfigured is one way of marking the addresses in that range as illegal or nonexistent with respect to the loading process. Memory configurations other than the default must be specified explicitly.

Unless otherwise specified, all discussion in this chapter of memory, addresses, and so on, concerns the configured sections of the address space.

Sections

A *section* of an object file is the smallest unit of relocation and must be a contiguous block of memory. You can identify a section with a starting address and a size. Information describing all the sections in a file is stored in *section headers* at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there can be *holes* or gaps between input sections (and between output sections), storage is allocated contiguously within each output section and cannot overlap a hole in memory.

Addresses

The *physical address* of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is relative to address zero of the virtual space and the system performs another address translation.

Binding

You might need to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called *binding,* and the section in question is said to be "bound to" or "bound at" the required address. While binding is most commonly relevant to output sections, you also can bind global symbols with an assignment statement in the `ld` command language.

Object files

Object files are produced both by the assembler (typically as a result of invoking the compiler) and by `ld`. `ld` accepts relocatable object files as input and produces an output object file that might or might not be relocatable. Under special circumstances, the input object files given to `ld` also can be absolute files. (See the section "Nonrelocatable Input Files" later in this chapter for details.)

Files produced by the compiler or assembler always contain three sections:

`.text`	contains the instruction text (for example, executable instructions)
`.data`	contains initialized data variables
`.bss`	contains uninitialized data variables

Files using shared libraries contain two additional sections:

`.init`	contains shared-library initialization fragments
`.lib`	contains the pathname to the shared library (for files using shared-library executable files)

Files calling shared-library executable files also contain dummy sections corresponding to the sections of the shared object file. For additional information, see Chapter 7, "Shared Libraries."

Here is an example of a typical (nonshared library) C program. If the source contained the following global declarations (not declared inside a function),

```
int i = 100;
char abc[200];
```

and the assignment,

```
abc[i] = 0;
```

compiled code from the C assignment would be stored in `.text`, the variable `i` would be located in `.data`, and `abc` would be located in `.bss`.

There is an exception, however, to this rule: Both initialized and uninitialized statics are allocated to the `.data` section (the value of an uninitialized static in a `.data` section is zero).

Options

You can intersperse options with filenames both on the command line and in an i-file. The ordering of options is not significant, except for the `-l` and `-L` options for specifying libraries.

The `-l` option is shorthand notation for specifying an archive library, which is a collection of object files. Thus, as is the case with any object file, libraries are searched as they are encountered. The `-L` option specifies an alternative directory for searching for libraries. To be effective, an `-L` option must, therefore, appear before any `-l` options.

All options for `ld` must be preceded by a hyphen (`-`), whether in the i-file or on the `ld` command line. Options that have an argument (except for the `-l` and `-L` options) are separated from the argument by white space (blanks or tabs). Table 15-1 lists the supported options.

Table 15-1 `ld` options

Option	Description
`-A`*factor*	Expands the default symbol table by the factor given.
`-F`	Performs the alignment necessary for demand paging. (Sections are aligned on stricter boundaries in the address space. Sections are blocked in the output file so that they begin on file-system block boundaries. In addition, the magic number 0413 is stored in the file header.)
`-L`*dir*	Changes the algorithm for searching for libraries to look in *dir* before looking in the default location. (This option is used for `ld` libraries in the same way the `-I` option is for compiler `#include` files. The `-L` option is useful for finding libraries that are not in the standard library directory. To be useful, though, this option must appear before the `-l` option.)
`-M`	Prints a warning message for all external variables that are multiply defined.
`-N`	Adjusts the load point of the data section so that it immediately follows the text section when loaded and stores the magic number 0407 in the header. (This prevents the text from being shared, which is the default.)
`-S`	Requests a silent `ld` run. (All error messages from errors that do not immediately stop the `ld` run are suppressed.)
`-V`	Prints, on the standard error output, a *version id* identifying the version of `ld` involved.
`-VS`*num*	Takes *num* as a decimal version number identifying the `a.out` file that is produced. (The version stamp is stored in the system header. This option is not directly recognized by the compiler [`cc`], so you must use the `-W` option to pass the version number to the link editor; for example, `-Wl,-VS` *num* where `-W` is an option to `cc` allowing arguments to be passed, `l` stands for the link editor [the destination of the argument], and `-VS` *num* are the arguments to `ld` that set the version number for the `a.out` file. Note that the space between `-VS` and *num* is required.)
`-e`*ss*	Defines the primary entry point of the output file to be the symbol given by the argument *ss*.
`-f`*bb*	Sets the default fill value. (The argument *bb* is a 2-byte constant. This value is used to fill holes formed within output sections. It is also used to initialize input `.bss` sections when they are combined with other non `.bss` input sections. If you don't use the `-f` option, the default fill value is zero for all sections except the `.tv` section, whose default fill value is 0xFFFF.)
`-ild`	Generates the sections reserved for use by the incremental link editor. (`-ild` invokes the `-r` option.)
`-l`*file*	Specifies an archive library file as `ld` input. (The argument *file* is a character string [less than ten characters] immediately following the `-l` without any intervening white space. As an example, `-lc` refers to `libc.a`, `-lC to libC.a`, and so on. The given archive library must contain valid object files as its members. The directory searched defaults to `/usr/lib`, finding `/usr/lib/libc.a`, `/usr/lib/libC.a`, and so on. [See also the `-L` option.])
`-m`	Produces a map or list of the input/output sections (including holes) on the standard output.
`-o`*nn*	Names the output object file. (The argument *nn* is the name of the A/UX system file to be used as the output file. The default output object filename is `a.out`. The option *nn* can be a full or partial A/UX pathname.)

Table 15-1 `ld` options *(continued)*

Option	Description
`-r`	Retains relocation entries in the output object file. (Relocation entries must be saved if the output file is to be used as an input file in a subsequent `ld` call. If the `-r` option is used, unresolved references do not prevent the creation of an output object file [such a file is not executable, of course]).
`-s`	Strips line number entries and symbol table information from the output object file. (Because relocation entries [`-r` option] are meaningless without the symbol table, you cannot use `-r` if you use `-s`. All symbols are stripped, including global and undefined symbols.)
`-t`	Disables checking of all instances of a multiply defined symbol to be sure they are the same size.
`-u`*sym*	Introduces an unresolved external symbol into the output file symbol table. (The argument *sym* is the name of the symbol. This option is useful for loading entirely from a library, since the symbol table is initially empty and an unresolved reference is needed to force the loading of an initial routine from the library.)
`-x`	Does not preserve any local (nonglobal) symbols in the output symbol table; enter external and static symbols only. (This option saves some space in the output file.)
`-z`	Catches references through `NULL` pointers. (The `z` is a mnemonic for "Do not place anything in address 0." This option is overridden if any section or memory directives are used.)

The `ld` command language

The command language of `ld` allows you control over all phases of the loading process. Typically, `ld` operates on files created by `as` without needing your intervention. However, you can write your own program specifying how `ld` is to manipulate the components of object files.

Input to `ld` is a series of directives that together have the effect of combining various reloctable input object files, binding all objects to known addresses, and resolving object references so that the resulting output object file is self-consistent and executable.

Expressions

Expressions can contain global symbols, constants, and most of the basic C language operators (see the last section of this chapter, "Syntax Diagram for Input Directives"). Constants in `ld` are defined as in C, with a number recognized as decimal unless preceded with `0` for octal or `0x` for hexadecimal.

◆ **Note** All numbers are treated as type `long int`. ◆

Symbol names can contain uppercase or lowercase letters, digits, and the underscore (_). Symbols within an expression have the value of the address of the symbol only. `ld` does not perform a symbol table look-up to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, and anything other than the value of the address.

`ld` uses a `lex`-generated input scanner to identify symbols, numbers, operators, and other elements. The current scanner design makes the following names reserved and unavailable as symbol or section names:

```
ALIGN       LENGTH      RANGE
ASSIGN      MEMORY      SECTIONS
BLOCK       NOLOAD      SPARE
DSECT       ORIGIN      TV
GROUP       PHY

align       len         phy
assign      length      range
block       o           spare
group       org
l           origin
```

Supported operators are shown in order of precedence in Table 15-2. These operators have the same meaning as in the C language. Precedence decreases from the top to the bottom of the table. Operators on the same line have the same precedence.

Table 15-2 Precedence of operators

Operator symbols					
`!`	`+`	`-` (unary minus)			
`*`	`/`	`%`			
`+`	`-` (binary minus)				
`>>`	`<<`				
`==`	`!=`	`>`	`<`	`<=`	`>=`
`&`					
`|`					
`&&`					
`||`					
`=`	`+=`	`-=`	`*=`	`/=`	

Assignment statements

External symbols can be defined and assigned addresses through the *assignment statement*. The syntax of the assignment statement is

symbol = *expression;*

or

symbol op = *expression;*

where *op* is one of the operators `+`, `-`, `*`, or `/`.

◆ **Note** Assignment statements must terminate with a semicolon. ◆

All assignment statements (with one exception, described in the following paragraph) are evaluated after allocation is performed; therefore, evaluation occurs after all input file-defined symbols are appropriately relocated but before the actual relocation of the text and data. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References to symbols given a value through an assignment statement within text and data access this latest assigned value. Assignment statements are processed in the same order in which they are input to `ld`.

Assignment statements are normally placed outside the scope of any section-definition directives. (See the section "Section Definition Directives," later in this chapter.) However, there is a special symbol, a *dot* (`.`), that can occur only within a section-definition directive. This symbol refers to the current address of the `ld` location counter. Thus, assignment expressions involving a dot are evaluated during the allocation phase of `ld`.

Assigning a value to the dot (`.`) symbol within a section-definition directive increments or resets the `ld` location counter and can create holes within the section (as described in "Section Definition Directives," later in this chapter).

Assigning the value of dot (`.`) to a conventional symbol permits the final allocated address of a particular point within the load run to be saved.

`align` is provided as a shorthand notation to allow you to align a symbol to an *n*-byte boundary within an output section, where *n* is a power of 2. For example, the expression

```
align(n)
```

is equivalent to

```
(. + n - 1) & (n - 1)
```

Link editor expressions can have either an absolute or a relocatable value, corresponding to a type of absolute or relocatable. When `ld` creates a symbol through an assignment statement, the symbol value takes on the type of the expression. That type depends on the following rules:

- An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable. The value is in relation to the section of the referenced symbol.
- All other expressions have absolute values.

Specifying a memory configuration

`MEMORY` directives are used to specify the following directives:

- the total size of the virtual space of an A/UX system
- the configured and unconfigured areas of the virtual space

If you do not supply any directives, `ld` assumes that all memory is configured. Macintosh computers that run A/UX have a minimum of 4 MB of RAM and use a virtual memory space of 4 GB. Generally, the only reason you would want to specify `MEMORY` directives for an A/UX application is to run it explicitly in physical, rather than virtual, memory space.

Using `MEMORY` directives, you can assign an arbitrary name of up to eight characters to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specified memory areas. Memory names can contain uppercase or lowercase letters, digits, and the three special characters dollar sign (`$`), dot (`.`), or underscore (`_`). Names of memory ranges are used only by `ld` and are not carried in the output file symbol table or headers.

◆ **Note** When you use `MEMORY` directives, all virtual memory that is not described in a `MEMORY` directive is considered to be unconfigured. `ld` does not use unconfigured memory in the allocation process; hence, nothing can be loaded, bound, or assigned to an address within unconfigured memory. ◆

As an option on the `MEMORY` directive, you can associate *attributes* with a named memory area. Doing so restricts the memory areas (with specific attributes) to which an output section can be bound. The attributes you assign to output sections are recorded in the appropriate section headers in the output file to allow for possible error checking in the future. For example, putting a text section into writable memory is one potential error condition. Currently, error checking of this type is not implemented.

The attributes currently accepted are as follows:

`R`	readable memory
`W`	writable memory
`X`	executable (instructions can reside in this memory)
`I`	initializable (stack areas are typically not initialized)

Other attributes can be added in the future, if necessary. If you do not specify any attributes in a `MEMORY` directive or if you do not supply any `MEMORY` directives, memory areas assume all of the attributes of `W`, `R`, `I`, and `X`.

The syntax of the `MEMORY` directive is

```
MEMORY
{
  name (attr): origin = virt-addr[,] length = mem-length
  ...
}
```

The keyword `origin` (or `org` or `o`) must precede the origin of a memory range and `length` (or `len` or `l`) must precede the length, as shown in the preceding prototype. The `origin` operand refers to the virtual address of the memory range. `origin` and `length` are entered as long integer constants in decimal, octal, or hexadecimal (standard C syntax). Origin and length specifications, as well as individual `MEMORY` directives, can be separated by white space or a comma.

By specifying `MEMORY` directives, you can tell `ld` that memory is configured in some manner other than the default. For example, if you need to prevent anything from being loaded to the first 0x10000 words of memory, you can do so with a `MEMORY` directive:

```
MEMORY
{
  valid : org = 0x10000, len = 0xFE0000
}
```

Region directives

This implementation of A/UX does not support region specifications, which are usually used only when developing UNIX kernels.

Section definition directives

You can use the `SECTIONS` directive to describe how input sections are to be combined, to direct where output sections should be placed (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections. Sections in `ld` are equivalent to segments in `as`.

In the default case (where no `SECTIONS` directives are given), all input sections of the same name appear in an output section of that name. For example, if a number of object files from the compiler are loaded, each containing the three sections `.text`, `.data`, and `.bss`, the output object file also contains the three sections `.text`, `.data`, and `.bss`. If two object files are loaded, one containing sections *s1* and *s2* and the other containing sections *s3* and *s4*, the output object file contains the four sections *s1*, *s2*, *s3*, and *s4*. The order of these sections depends on the order in which the link editor sees the input files.

The basic syntax of the `SECTIONS` directive is

```
SECTIONS
{
        secname :
        {
                file-specification ...,
                assignment-statement ...
        }
          ...
}
```

The various types of section definition directives are discussed in the remainder of this section.

File specifications

Within a section definition, the files and file sections to be included in the output section are listed in the order in which they are to appear. Sections from an input file are specified by

filename `(` *secname* `...)`

Sections of an input file are separated by white space or commas (or both), as are the file specifications themselves.

If a filename appears with no sections listed, all sections from the file are loaded into the current output section; for example,

```
SECTIONS
{
        outsec1:
        {
                file1.o  (sec1)
                file2.o
                file3.o  (sec1,  sec2)
        }
}
```

The order in which the input sections appear in the output section *outsec1* is given as follows:

1. Section *sec1* from file *file1*`.o`.
2. All sections from *file2*`.o`, in the order they appear in the file.
3. Section *sec1* from file *file3*`.o`, then section *sec2* from file *file3*`.o`.

If any additional input files contain input sections named *outsec1*, these sections are loaded following the last section named in the *outsec1* definition. If there are any other input sections in *file1*`.o` or *file3*`.o`, they are placed in output sections with the same names as the input sections.

Loading a section at a specified address

You might want to bond an output section to a specific virtual address to take advantage of a particular paging efficiency. This can be done as shown in the following `SECTIONS` directive example.

```
SECTIONS
{
        outsec addr:
        {
                file-spec  (secname)
        }
        ...
}
```

addr is the bonding address, expressed as a C constant. If *outsec* does not fit at *addr* (perhaps because of holes in the memory configuration or because *outsec* is too large to fit without overlapping some other output section), `ld` issues an appropriate error message.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The `SECTIONS` directives that define output sections do not have to be given to `ld` in any particular order.

`ld` does not ensure that the size of each section consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by four. Although it is not recommended, you can use the `ld` directives to force a section to start on an odd byte boundary, if unforeseen circumstances force you into this solution. If a section starts on an odd byte boundary, the section contents either are accessed incorrectly or are not executed properly. If you specify an odd byte boundary, `ld` issues a warning message.

Aligning an output section

You can request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The `ALIGN` option of the `SECTIONS` directive performs this function, so that the option

```
ALIGN(n)
```

is equivalent to specifying a bonding address of

```
( . + n - 1) & (n - 1)
```

◆ **Note** This `ALIGN` option is different from the `align` option discussed in the section "Assignment Statements," earlier in this chapter. `ALIGN` binds sections to an address boundary, while `align` binds a specific object to an address boundary. ◆

You should note that the `as` assembler always pads the sections it generates to a full word length, making explicit alignment specifications unnecessary. This also holds true for the compilers `c89` and `cc`. Here is an example of section alignment:

```
SECTIONS
{
          outsec  ALIGN(0x20000):
          {
               file-spec (secname)
          }
          . . .
}
```

The output section *outsec* is not bound to any given address, but is loaded to some virtual address that is a multiple of 0x20000 (for example, at address 0x0, 0x20000, 0x40000, 0x60000, and so on).

The default section alignment action for `ld` on MC68000-family systems is to align the code (`.text`) at byte 0 and the data (`.data` and `.bss` combined) at the 4 MB boundary (byte 10487576). Since MMU requirements vary from system to system, alignment is not always desirable. The version of `ld` for MC68000-family systems, therefore, provides a mechanism to allow the specification of different section alignments for each system, allowing you to align each section separately on *n*-byte boundaries, where *n* is a multiple of 512.

The default allocation algorithm for `ld` is as follows:

1. Load all input `.text` sections together into one output section. This output section is called `.text` and is bound to an address of 0x0.
2. Load all input `.data` sections together into one output section. This output section is called `.data` and is bound to an address aligned to a machine-dependent constant.
3. Load all input `.bss` sections together into one output section. This output section is called `.bss` and is allocated so as to follow the output section `.data` immediately. Note that the output section `.bss` is not given any particular address alignment.

Specifying any `SECTIONS` directives results in this default allocation not being performed.

When all input files are processed (and if no override is provided), `ld` searches the list of library directories (as with the `-l` flag option) for a file named `default.ld`. If this file is found, it is processed as an `ld` instruction file (or i-file). The `default.ld` file should specify the required alignment, as outlined in the following paragraphs. If it does not exist, the default alignment action is taken.

The `default.ld` file should appear as in the following example, with *align-value* replaced by the alignment requirement in bytes. The default allocation of `ld` is equivalent to supplying the following directive:

```
SECTIONS
{
    .text    : { }
    GROUP ALIGN(align-value) :
    {
         .data     : { }
         .bss      : { }
    }
}
```

In this example, the variable *align-value* is a machine-dependent constant.

◆ **Note** The current (MC68000 family) system requires a *data rounding* of 2 MB. This requirement is subject to change as systems evolve. ◆

The `GROUP` directive ensures that the two output sections, `.data` and `.bss`, are allocated (grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If you want to place `.text`, `.data`, and `.bss` in the same segment, you should use the following `SECTIONS` directive.

```
SECTIONS
{
    GROUP           :
    {
         .text      : { }
         .data      : { }
         .bss       : { }
    }
}
```

Note that there are still three output sections (`.text`, `.data`, and `.bss`), but they are now allocated into consecutive virtual memory.

This entire group of output sections can be bound to a starting address or aligned simply by adding a field to the `GROUP` directive. To bind to 0xC0000, use

```
GROUP 0xC0000: {
```

To align to 0x10000, use

```
GROUP ALIGN(0x10000): {
```

With this addition, first the output section `.text` is bound at 0xC0000 (or is aligned to 0x10000); the remaining members of the group are allocated into the next available memory locations in order of their appearance.

When the `GROUP` directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
    .text: { }
    .data ALIGN(0x20000): { }
    .bss: { }
}
```

The `.text` section starts at virtual address 0x0 and the `.data` section at a virtual address aligned to 0x20000. The `.bss` section follows immediately after the `.text` section, but only if there is enough space. If there is not enough space, it follows the `.data` section.

The order in which output sections are defined to `ld` cannot be used to force a certain allocation order in the output file.

Files that need to load in a shared library have the `.init` and `.text` sections grouped together. In the final stage of loading, the `.init` section becomes part of the `.text` section.

Creating holes within output sections

The special dot (`.`) symbol appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, it causes the `ld` location counter to be incremented or reset, and a hole is left in the output section.

Holes that are built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character, 0x00, or a supplied fill character). See the definition of the `-f` option in the section "Options" and the discussion of filling holes in the section "Initialized Section Holes or `.bss` Sections" in this chapter.

Consider the following section definition:

```
SECTIONS
{
outsec:
    {
        . += 0x1000;
        f1.o (.text)
        . += 0x100;
        f2.o (.text)
        . = align (4);
        f3.o (.text)
    }
}
```

The effect of this command is as follows:

1. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input file *f1*.`o(.text)` is loaded after this hole.
2. The text of input file *f2*`.o` begins at 0x100 bytes following the end of *f1*`.o(.text)`.
3. The text of *f3*`.o` is loaded to start at the next full word boundary following the text of *f2*`.o` with respect to the beginning of *outsec*.

For the purposes of allocating and aligning addresses within an output section, `ld` treats the output section as if it began at address zero. If, in this example, *outsec* is ultimately loaded to start at an odd address, the part of *outsec* built from *f3*`.o(.text)` also starts at an odd address, even though *f3*`.o(.text)` is aligned to a full word boundary. You can prevent this result by specifying an alignment factor for the entire output section:

```
outsec ALIGN(4): {
```

Expressions that decrement the dot (`.`) symbol are illegal. For example, subtracting a value from the location counter is not allowed, since overwrites are not allowed. The most common operators in expressions that assign a value to the dot (`.`) are `+=` and `align`.

Creating and defining symbols at loading time

You can use the assignment instruction of `ld` to give symbols a loading-dependent value. Typically, there are three types of assignments:

- use of the dot (.) to adjust the `ld` location counter during allocation
- use of the dot (.) to assign an allocation-dependent value to a symbol
- assignment of an allocation-independent value to a symbol

The first case was discussed in the previous section. The second case provides a means to assign addresses (known only after allocation) to symbols; for example,

```
SECTIONS
{
    outsec1: {file-spec (secname)}
    outsec2:
    {
        file1.o (s1)
        s2_start = . ;
        file2.o (s2)
        s2_end = . - 1;
    }
}
```

The symbol *s2*_`start` is defined to be the address of *file2*.`o(`*s2*`)`, and *s2*_`end` is the address of the last byte of *file2*.`o(`*s2*`)`. Consider the following example:

```
SECTIONS
{
        outsec1:
        {
                file1.o (.data)
                mark = .;
                . += 4;
                file2.o (.data)
        }
}
```

In this example, the symbol `mark` is created and is equal to the address of the first byte beyond the end of the *file1*.`o .data` section. Four bytes are reserved for a future run-time initialization of the symbol `mark`. The type of the symbol is a long integer (32 bits).

Assignment instructions involving the dot (`.`) symbol must appear within `SECTIONS` definitions, since they are evaluated during allocation. Assignment instructions that do not involve the dot (`.`) symbol can appear within `SECTIONS` definitions, but typically do not. Such instructions are evaluated after allocation is complete.

It is risky to reassign a defined symbol to a different address. For example, if a symbol within `.data` is defined, initialized, and referred to within a set of object files being loaded, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. The associated initialized data are not moved to the new address. `ld` issues warning messages for each defined symbol that is being redefined within an i-file. Assignments of absolute values to new symbols are safe, however, because there are no references or initialized data associated with the symbol.

Allocating a section into named memory

The link editor provides a mechanism for allowing you to specify a section to be loaded somewhere within a specific, named memory area (as previously specified on a `MEMORY` directive) using the > operator. The `>` notation is borrowed from the UNIX system concept of redirected output. Consider the following example:

```
MEMORY
{
    mem1:           o=0x000000    l=0x10000
    mem2 (RW):      o=0x020000    l=0x40000
    mem3 (RW):      o=0x070000    l=0x40000
    mem1:           o=0x120000    l=0x04000
}

SECTIONS
{
    outsec1: {f1.o(.data) } > mem1
    outsec2: {f2.o(.data) } > mem3
}
```

This code fragment directs `ld` to place *outsec1* at the first location within the memory area named *mem1* that is large enough to hold the section (somewhere within the address range `0x0-0xFFFF` or `0x120000-0x123FF`). The *outsec2* is to be placed similarly in the address range `0x70000-0xAFFFF`.

Initialized section holes or `.bss` sections

When holes are created within a section (as in the example in the section "Creating Holes Within Output Sections," earlier in this chapter), `ld` normally puts out bytes of zero as *fill*. By default, `.bss` sections are not initialized at all; that is, no initialized data, not even zeros, are generated for any `.bss` section by the assembler or supplied by the link editor.

You can use initialization options in a `SECTIONS` directive to set such holes or to set `.bss` sections to an arbitrary 2-byte pattern.

◆ **Note** Such initialization options apply only to `.bss` sections or holes. ◆

In an application, for example, you might want an uninitialized data table to be initialized to a constant value without recompiling the `.o` file or filling a hole in the text area with a transfer to an error routine. You can designate that either specific areas within an output section or the entire output be initialized. Because no text is generated for an uninitialized `.bss` section, however, the entire section is initialized if part of such a section is initialized.

In other words, if a `.bss` section is to be combined with a `.text` or `.data` section (both of which are initialized), or if part of an output `.bss` section is to be initialized, one of the following conditions holds:

- Explicit initialization options must be used to initialize all `.bss` sections in the output section.
- `ld` uses the default fill value to initialize all `.bss` sections in the output section.

Consider the following `ld` i-file:

```
SECTIONS
{
    sec1:
    {
        f1.o (.text)
        . += 0x200;
        f2.o (.text)
    } = 0xDFFF
    sec2:
    {
        f1.o (.bss)
        f2.o (.bss)
    } = 0x1234
    sec3:
    {
        f3.o (.bss)
        ...
    } = 0xFFFF
    sec4: {f4.o (.bss)}
}
```

In the preceding example, the 0x200 byte hole in section *sec1* is filled with the value `0xDFFF`. In section *sec2*, *f1*`.o (.bss)` is initialized to the default fill value of `0x00` and *f2*`.o (.bss)` is initialized to `0x1234`. All `.bss` sections within *sec3*, as well as all holes, are initialized to `0xFFFF`. Section *sec4* is not initialized; that is, no data are written to the object file for this section.

Notes and special considerations

The following sections are collections of additional information that are helpful in understanding the link editor.

Using archive libraries

Each member of an archive library (for example, `libc.a`) is a complete object file, typically consisting of the standard three sections:

- `.text`
- `.data`
- `.bss`

Shared-library archives contain one or two additional sections:

- `.init`
- `.lib`

In addition to these sections, files calling on shared-library executable files contain dummy sections corresponding to sections of the shared object. For further information, see Chapter 7, "Shared Libraries."

Archive libraries are created through the use of the A/UX system `ar` command on object files generated by running `cc` or `as`. Shared libraries are created using the `mkshlib` command. An archive library is always processed using *selective inclusion:* Only those members that resolve existing undefined-symbol references are taken from the library for loading.

Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for loading whenever the following conditions exist:

- A reference to a symbol is defined in that member.
- The reference is found by `ld` prior to the actual scanning of the library.

When a library member is included by searching the library inside a `SECTIONS` directive, all input sections from the member are included in the output section being defined.

When a library member is included by searching the library outside a `SECTIONS` directive, all input sections from the member are included in the output section with the same name. That is, the `.text` section of the member goes into the output section named `.text`, the `.data` section of the member goes into `.data`, the `.bss` section of the member goes into `.bss`, and so on. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, the following conditions:

- Specific members of a library cannot be referred explicitly in an i-file.
- The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

The `-l` option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. They do not, however, have to be. Furthermore, you can specify archive libraries without using the `-l` option simply by giving the full or relative A/UX system pathname.

◆ **Note** The ordering of archive libraries is important, because a member extracted from the library must satisfy a reference that is known to be unresolved at the time the library is searched. ◆

You can specify archive libraries more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. `ld` cycles through this symbol table until it determines that it cannot resolve any more references from that library.

`ld` uses a random-access library. All machines running pre-System V UNIX use an old format library that must be searched linearly.

The link editor makes one search through a library in the old format, but continues to search through a library in the new format until it determines that it can resolve no more references from that library. Because of the different searching algorithms used, programs that are ported from pre–System V UNIX machines can include files from libraries in a different order.

Be careful when using archive libraries in a subsystem loading environment. If a member of an archive (an object file) is to be included in a subsystem final load file, there must be a reference within the subsystem being loaded to a symbol defined in that object file. You can use the `-u` option to create unresolved references that force the loading of archive members.

Consider the following example:

- The input files *file1*`.o` and *file2*`.o` each contain a reference to the external function `FCN`.
- Input *file1*`.o` contains a reference to symbol `ABC`.
- Input *file2*`.o` contains a reference to symbol `XYZ`.
- Library `liba.a`, member 0, contains a definition of `XYZ`.
- Library `libc.a`, member 0, contains a definition of `ABC`.
- Both libraries have a member 1 that defines `FCN`.

Depending on the order in which files and libraries appear on the command line, different library members can be included for loading. If the `ld` command is entered as

```
ld file1.o -la file2.o -lc
```

the `FCN` references are satisfied by `liba.a`, member 1; `ABC` is obtained from `libc.a`, member 0; and `XYZ` remains undefined (because the library `liba.a` is searched before *file2*`.o` is specified). If the `ld` command is entered as

```
ld file1.o file2.o -la -lc
```

the `FCN` references are satisfied by `liba.a`, member 1; `ABC` is obtained from `libc.a`, member 0; and `XYZ` is obtained from `liba.a`, member 0. If the `ld` command is entered as

```
ld file1.o file2.o -lc -la
```

the `FCN` references are satisfied by `libc.a`, member 1; `ABC` is obtained from `libc.a`, member 0; and `XYZ` is obtained from `liba.a`, member 0.

You can use the `-u` option to force the loading of library members when the loading run does not contain an actual external reference to the members. For example,

```
ld -u rout1 -la
```

creates an undefined symbol called `rout1` in the `ld` global symbol table. If any member of library `liba.a` defines this symbol, that member is extracted. Without the `-u` option, there would have been no trigger to cause `ld` to search the archive library.

Dealing with holes in physical memory

When memory configurations are defined so that unconfigured areas exist in virtual memory, each application or user has the responsibility to form output sections that fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
    mem1:    o = 0x00000    l = 0x02000
    mem2:    o = 0x40000    l = 0x05000
    mem3:    o = 0x20000    l = 0x10000
}
```

Let the files *f1*`.o`, *f2*`.o`, … *fn*`.o` each contain the standard three sections `.text`, `.data`, and `.bss`, and let the combined `.text` section be 0x12000 bytes. There is no configured area of memory into which this section can be placed. Appropriate directives must be supplied to break up the `.text` output section so that `ld` can perform allocation. For example,

```
SECTIONS
{
    txt1:
    {
        f1.o (.text)
        f2.o (.text)
        f3.o (.text)
    }
```

(continued)➡

```
    txt2:
    {
        f4.o (.text)
        f5.o (.text)
        f6.o (.text)
    }
    . . .
}
```

Allocation algorithm

An output section is formed either as a result of a `SECTIONS` directive or by combining input sections of the same name. An output section can be made up of zero or more input sections. After the composition of an output section is determined, it must be allocated into configured virtual memory. `ld` uses an algorithm that attempts to minimize fragmentation of memory, which increases the possibility that a loading run can allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. It allocates any output sections for which explicit bonding addresses are specified.
2. It allocates any output sections to be included in a specified memory area. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory area, taking into consideration any alignment.
3. It allocates output sections that are not handled by steps 1 or 2.

If all memory is contiguous and configured (the default), and no `SECTIONS` directives are given, output sections are allocated in the order they appear to `ld`, normally `.text`, `.data`, `.bss`. Otherwise, output sections are allocated, in the order they were defined or made known to `ld`, into the first available space in which they fit.

Incremental loading

As previously mentioned, the output of `ld` can be used as an input file to subsequent `ld` runs, provided that the relocation information is retained (using the `-r` option). With large applications you might find it desirable to partition C programs into subsystems, load each subsystem independently, and then load the entire application. For example,

Step 1:

```
    ld -r  -o outfile1  i-file1

    /* i-file1  */
    SECTIONS
    {
        ss1:
        {
            f1.o
            f2.o
            ...
            fn.o
        }
    }
```

Step 2:

```
    ld -r  -o outfile2  i-file2

    /* i-file2  */
    SECTIONS
    {
        ss2:
        {
            g1.o
            g2.o
            ...
            gn.o
        }
    }
```

Step 3:

```
ld  -a -m -o final.out outfile1 outfile2
```

By judiciously forming subsystems, applications can achieve a form of *incremental loading,* whereby it is necessary to reload only a portion of the total load when a few programs are recompiled. To apply this technique, follow two simple rules:

1. Intermediate loads must contain only `SECTIONS` declarations and be concerned only with the formation of output sections from input files and input sections. You must not do any binding of output sections in these runs.
2. All allocation and memory directives, as well as any assignment statements, must be included only in the final `ld` call.

`DSECT`, `COPY`, and `NOLOAD` sections

You can give sections a type in a section definition.

The `DSECT` option creates a *dummy section,* which has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it does not take up memory and does not show up in the memory map (the `-m` option) generated by `ld`.
2. It can overlay other output sections and even unconfigured memory. Dummy sections can overlay other dummy sections.
3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file symbol table with the same value they would have if the dummy section were actually loaded at its virtual address. Other input sections can reference `DSECT`-defined symbols. Undefined external symbols found within a dummy section cause specified archive libraries to be searched; any members that define such symbols are loaded normally (not in the dummy section or as a dummy section).
4. None of the section contents, relocation information, or line-number information associated with the section is written to the output file.

A *copy section* is created by the `COPY` option. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information are written to the output file.

A *noload section* is allocated virtual space, appears in the memory map, and so forth. A section of the type `NOLOAD` differs from a normal output section in that text and data are not written to the output file. For example,

```
SECTIONS
{
    name1 0x200000 (DSECT)    : {file1.o}
    name2 0x400000 (COPY)     : {file2.o}
    name3 0x600000 (NOLOAD)   : {file3.o}
}
```

Here, none of the sections from *file1*`.o` are allocated, but all symbols are relocated as though the sections were loaded at the specified address. Other sections can refer to any of the global symbols and are resolved correctly.

Output file blocking

You can use two options to affect the physical file offsets of the information written to the output file by `ld`:

- The `BLOCK` option permits any output section to be aligned in the output field at a specified *n*-byte boundary.
- The `-B` option causes padding sections to be generated in the output file.

Both features are provided explicitly for the use of `ldp`, which constructs *pfile*. The output sections of a *pfile* have certain requirements in terms of physical file offsets. These requirements can be met by using `BLOCK` and `-B`.

You can apply the `BLOCK` option to any output section or `GROUP` directive. It directs `ld` to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated or on any part of the loading process. It is used purely to adjust the physical position of the section in the output file:

```
SECTIONS
{
.text BLOCK(0x200):{ }
    .data ALIGN(0x20000)BLOCK(0x200):{ }
}
```

In this `SECTIONS` directive example, `ld` ensures that each section, `.text` and `.data`, is physically written at a file offset that is a multiple of 0x200 (for example, at an offset of 0, 0x200, 0x400, ..., and so on, in the file).

Nonrelocatable input files

If you intend to use a file produced by `ld` in a subsequent `ld` run, you should set the `-r` option for the first `ld` run. This preserves relocation information and permits the sections of the file to be relocated by the subsequent `ld` run.

When `ld` detects an input file that does not have relocation or symbol table information, it gives a warning message. Such information can be removed by `ld` (see the `-s` option in the section "Options," earlier in this chapter) or by the `strip`(1) program. Note, however, that the loading run continues, using the nonrelocatable input file. For such a load to be successful (that is, actually and correctly loading all input files, relocating all symbols, resolving unresolved references, and so on), two conditions for the nonrelocatable input files must be met:

1. Each input file must not have unresolved external references.
2. Each input file must be bound to the same virtual address as it was in the `ld` run that created it.

Note that if these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this restriction, you must take extreme care when supplying such input files to `ld`.

The `-ild` option

When the `-ild` option is used, the link editor creates a pair of dummy sections of type `DSECT` for each unallocated, configured area of memory. These dummy sections have unique names in the form of `.i_l_d`*nn*, where *nn* is a 2-digit decimal integer in the range from 00 through 99. At most, 50 pairs of these sections are created by the link editor. These sections identify the boundaries of the unused memory space and are similar to `.bss` sections in that they do not contain any text or initialized data. The link editor also creates a dummy section named `.history`. These sections are used later by the incremental link editor.

Error messages

The following sections report the error messages you can receive from `ld`. The sections are arranged by general topic.

Corrupt input files

Certain error messages indicate that the input file is corrupt, nonexistent, or unreadable. If you receive any of them, you should check that the file is in the correct directory with the correct permissions. If the object file is corrupt, try recompiling or reassembling it. The error messages are as follows:

`Can't open` *name*`.`

`Can't read archive header from archive` *name*`.`

`Can't read file header of archive` *name*`.`

`Can't read 1st word of file` *name*`.`

`Can't seek to the beginning of file` *name*`.`

`Fail to read file header of` *name*`.`

`Fail to read lnno of section` *sect* `of file` *name*`.`

`Fail to read magic number of file` *name*`.`

`Fail to read section headers of file` *name*`.`

`Fail to read section headers of library` *name* `member` *number*`.`

`Fail to read symbol table of file` *name*`.`

`Fail to read symbol table when searching libraries.`

`Fail to read the aux entry of file` *name*`.`

`Fail to read the field to be relocated.`

`Fail to seek to symbol table of file` *name*`.`

`Fail to seek to symbol table when searching libraries.`

`Fail to seek to the end of library` *name* `member` *number*`.`

`Fail to skip aux entries when searching libraries.`

`Fail to skip the mem of struct of` *name*.

`Illegal relocation type.`

`No reloc entry found for symbol.`

`Reloc entries out of order in section` *sect* `of file` *name*.

`Seek to` *name* `section` *sect* `failed.`

`Seek to` *name* `section` *sect* `lnno failed.`

`Seek to` *name* `section` *sect* `reloc entries failed.`

`Seek to relocation entries for section` *sect* `in file` *name* `failed.`

Errors during output

Certain errors occur because `ld` cannot write to the output file. This usually indicates that the file system is out of space. These kinds of messages include the following:

`Cannot complete output file` *name*. `Write error.`

`Fail to copy the rest of section` *num* `of file` *name*.

`Fail to copy the bytes that need no reloc of section` *num* `of file.`

name `I/O error on output file` *name*.

Internal errors

Certain messages indicate that something is wrong with `ld` internally. If you receive them, there is probably nothing you can do except to obtain help from another experienced user of `ld`. Such messages are as follows:

`Attempt to free nonallocated memory.`

`Attempt to reinitialize the SDP aux space.`

`Attempt to reinitialize the SDP slot space.`

`Default allocation did not put .data and .bss into the same region.`

`Failed to close SDP symbol space.`

`Failure dumping an AIDFN`*xxx* `data structure.`

`Failure in closing SDP aux space.`

`Failure to initialize the SDP aux space.`

`Failure to initialize the SDP slot space.`

`Internal error: audit_groups, address mismatch.`

`Internal error: audit_group, finds a node failure.`

`Internal error: fail to seek to the member of` *name*`.`

`Internal error: in allocate lists, list confusion (`*num num*`).`

`Internal error: invalid aux table id.`

`Internal error: invalid symbol table id.`

`Internal error: negative aux table ld.`

`Internal error: negative symbol table id.`

`Internal error: no symtab entry for DOT.`

`Internal error: split_scns, size of` *sect* `exceeds its new displacement.`

Allocation errors

Certain error messages appear during the allocation phase of the load. They generally appear if a section or group does not fit at a certain address or if the given `MEMORY` or `SECTION` directives conflict in some way. If you are using an i-file and receive such messages, check that `MEMORY` and `SECTION` directives allow enough room for the sections to ensure that nothing overlaps and that nothing is being placed in unconfigured memory. For more information, see the sections "The `ld` Command

Language" and "Notes and Special Considerations," both earlier in this chapter. These messages are as follows:

`Bond address` *address* `for` *sect* `is not in configured memory.`

`Bond address` *address* `for` *sect* `overlays previously allocated section` *sect* `at` *address*`.`

`Can't allocate output section` *sect*`, of size` *num*`.`

`Can't allocate section` *sect* `into owner` *mem*`.`

`Default allocation failed:` *name* `is too large.`

`GROUP containing section` *sect* `is too big.`

`Memory types` *name1* `and` *name2* `overlap.`

`Output section` *sect* `not allocated into a region.`

sect `at` *address* `overlays previously allocated section` *sect* `at` *address*`.`

sect`, bonded at` *address*`, won't fit into configured memory.`

sect `enters unconfigured memory at` *address*`.`

`Section` *sect* `in file` *name* `is too big.`

Misuse of link editor directives

Certain error messages are explanations that occur following the misuse of an input directive. If you receive them, review the appropriate section in this chapter. These messages and brief explanations of their causes follow:

`Adding` *name*`(`*sect*`) to multiple output sections.`
The input section is mentioned twice in the `SECTIONS` directive.

`Bad attribute value in MEMORY directive: c.`
The attribute *c* is illegal. An attribute must be one of `R`, `W`, `X`, or `I`.

`Bad flag value in SECTIONS directive,` *option.*
Only the `-l` option is allowed inside a `SECTIONS` directive.

`Bad fill value.`
The fill value must be a 2-byte constant.

`Bonding excludes alignment.`
The section is bound at the given address, regardless of the alignment of that address.

`Cannot align a section within a group.`
`Cannot bond a section within a group.`
`Cannot specify an owner for sections within a group.`
The entire group is treated as one unit, so the group can be aligned or bound to an address, but the sections making up the group cannot be handled individually.

`DSECT` *sect* `can't be given an owner.`
`DSECT` *sect* `can't be linked to an attribute.`
Because dummy sections do not participate in the memory allocation, it is meaningless for a dummy section to be given an owner or an attribute.

`Region commands not allowed.`
The A/UX implementation of the link editor does not accept the `REGION` commands.

`Section` *sect* `not built.`
The most likely cause of this is a syntax error in the `SECTIONS` directive.

`Semicolon required after expression.`
`Statement ignored.`
This is caused by a syntax error in an expression.

`Usage of unimplemented syntax.`
The A/UX implementation of `ld` does not accept all possible commands.

Misuse of expressions

Certain errors arise from the misuse of an input expression. If you receive any of the following messages, please review the appropriate section in this chapter.

`Absolute symbol` *`name`* `being redefined.`
An absolute symbol cannot be redefined.

`ALIGN illegal in this context.`
Alignment of a symbol can be done only within a `SECTIONS` directive.

```
Attempt to decrement DOT.
Illegal assignment of physical address to DOT.
Illegal operator in expression.
Misuse of DOT symbol in assignment instruction.
```
You cannot use the dot (`.`) symbol in assignment statements that are outside of `SECTIONS` directives.

`Symbol` *`name`* `is undefined.`
All symbols referred to in an assignment statement must be defined.

`Symbol` *`name`* `from file name being redefined.`
A defined symbol cannot be redefined in an assignment statement.

`Undefined symbol in expression.`
All symbols used in expressions must be defined.

Misuse of options

Certain errors arise from the misuse of options. If you receive any of the following messages, please review the appropriate section of this book.

```
Both -r and -s flags are set.

-s flag turned off.

Can't find library libx.a.
```

`-L path too long (`*string*`).`

`-o file name too large (>128 char), truncated to (`*string*`).`

`Too many -L options, seven allowed.`

Some options require space before the argument, and some do not; see the section "Options," earlier in this chapter. Including extra space or not including the required space is the most likely cause of the following messages:

option `flag does not specify a number.`

option `is an invalid flag.`

`-e flag does not specify a legal symbol name:` *name*.

`-f flag does not specify a two-byte number:` *num*.

`No directory given with -L.`

`-o flag does not specify a valid file name:` *string*.

`-l flag (specifying a default library) is not supported.`

`-u flag does not specify a legal symbol name:` *name*.

Space constraints

Certain error messages can occur if `ld` attempts to allocate more space than is available. If you receive them, you should attempt to decrease the amount of space used by `ld`. You can do this by making the i-file less complicated or by using the `-r` option to create intermediate files. These space constraint messages are as follows:

`Fail to allocate` *num* `bytes for slotvec table.`

`Internal error: aux table overflow.`

`Internal error: symbol table overflow.`

`Memory allocation failure on` *num*`-byte call.`

`Memory allocation failure on realloc call.`

`Run is too large and complex.`

Miscellaneous errors

Errors occur for many reasons. If one occurs that is not explained in a previous section, refer to the error message for an indication of where to look in this reference. Miscellaneous error messages are as follows:

`Archive symbol table is empty in archive` *name*`, execute`
`'ar ts` *name*`' to restore archive symbol table.`
On systems with a random-access archive capability, such as A/UX, the link editor requires that all archives have a symbol table. This symbol table might have been removed by `strip`.

`Can't create intermediate ld file` *name*.
`Can't open internal file` *name*.
These two messages are possible only when the link editor uses two processes. They indicate that the temp directory (usually `/tmp` or `/usr/tmp`) is out of space, or that the link editor does not have permission to write in it.

`Cannot create output file` *name*.
You might not have write permission in the directory where the output file is to be written.

`File` *name* `is of unknown type, magic number =` *num*.
`Ifile nesting limit exceeded with file` *name*.
i-files can be nested 16 deep.

`Library` *name*, `member has no relocation information.`
`Multiply defined symbol` *sym*`, in` *name* `has more than one size.`
A multiply defined symbol is not defined in the same manner in all files.

name(sect) `not found.`
An input section specified in a `SECTIONS` directive was not found in the input file.

`Section` *sect* `starts on an odd byte boundary!`
This warning occurs only if you specifically bind a section at an odd boundary.

`Sections .text, .data or .bss not found;`
`Optional header may be useless.`
The system `a.out` header uses values found in the `.text`, `.data`, and `.bss` section headers.

`Line nbr entry (`*num num*`) found for nonrelocatable symbol:`
`Section` *sect*, `file` *name*.
This error is generally caused by an interaction of `yacc`(1) and `cc`(1). See the section "Notes and Special Considerations," earlier in this chapter.

`Undefined symbol` *sym* `first referenced in file` *name*.
Unless you use the `-r` option, `ld` requires that all referenced symbols be defined.

`Unexpected EOF (End Of File).`
There is a syntax error in the i-file.

Syntax diagram for input directives

Input to `ld` is a series of directives that together have the effect of combining various relocatable input object files, binding all objects to known addresses, and resolving object references so the resulting output object file is self-consistent and executable. Table 15-3 contains syntax diagrams for the input directives.

In Table 15-3, a particular notation is used. The terms on the left define the terms on the right. For example, the expansion

term	→	*directive1*
	→	*directive2*

means that *term* can be made up of *directive1* or *directive2*.

◆ **Note** Number suffixes have been added to some metalanguage terms to illustrate treatment of multiple arguments. You should ignore these suffixes when seeking the definition of such terms. ◆

Ellipses (...) indicate that several of the elements on the right can comprise a left-hand element. For example, the expansion

term	→	*directive* ...

means that *term* is made up of one or more *directives*. Brackets indicate optional directives, and braces indicate that the contents must be included in the directive.

For *flags*, one or more blanks, tabs, or newlines can be substituted wherever there is a space between a flag option and its argument.

Table 15-3 Directive expansion

Directive	→	**Expanded directive**
file	→	*cmd* ...
cmd	→	*memory*
	→	*sections*
	→	*assignment*
	→	*filename*
	→	*flags*
memory	→	`MEMORY` { *memory-spec1* [[,] *memory-spec2*]... }
memory-spec	→	*name* [*attributes*] : *origin-spec* [,] *length-spec*
attributes	→	([R][W][X][I])
origin-spec	→	*origin* = *long*
length-spec	→	*length* = *long*
origin	→	`ORIGIN`
	→	`o[rigin]`
	→	`o[rg]`
length	→	`LENGTH`
	→	`l[ength]`
	→	`l[en]`
sections	→	`SECTIONS` { *sec-or-group* ... }
sec-or-group	→	*section*
	→	*group*
	→	*library*
section	→	*name sec-options* : { *statement-list* } [*fill*] [*mem-spec*]
sec-options	→	[*addr*] [*align-option*] [*block-option*] [*type-option*]
addr	→	*long*
align-option	→	*align* (*long*)

Table 15-3 Directive expansion *(continued)*

Directive	→	**Expanded directive**
align	→	`ALIGN`
	→	`align`
block-option	→	*block* (*long*)
block	→	`BLOCK`
	→	`block`
type-option	→	`(DSECT)`
	→	`(NOLOAD)`
	→	`(COPY)`
statement-list	→	*statement1* [*statement2*] ...
statement	→	*filename* [(*name-list*)] [*fill*] *library assignment*
name-list	→	*name1* [[,] *name2*] ...
fill	→	= *long*
library	→	`-l` *name*
assignment	→	*lside assign-op expr end*
lside	→	*name*
	→	`.`
assign-op	→	`=`
	→	`+=`
	→	`-=`
	→	`*=`
	→	`/=`
expr	→	*term*
	→	*expr binary-op expr* *
term	→	*long*
	→	*name*
	→	*align* (*term*) *
	→	(*expr*)
	→	*unary-op term*
unary-op	→	`!`
	→	`-`

(continued)➡

Table 15-3 Directive expansion *(continued)*

Directive	→	**Expanded directive**
binary-op	→	`*`
	→	`/`
	→	`%`
	→	`+`
	→	`-`
	→	`>>`
	→	`<<`
	→	`==`
	→	`!=`
	→	`>`
	→	`<`
	→	`<=`
	→	`>=`
	→	`&`
	→	`|`
	→	`&&`
	→	`||`
end	→	;
	→	,
group	→	GROUP *group_options* : {*section-list*} [*mem-spec*]
group-options	→	[*addr*] [*align-option*]
section-list	→	*section1* [[, [*section2*] ...
mem-spec	→	> *name*
	→	> *attributes*

Table 15-3 Directive expansion *(continued)*

Directive	→	**Expanded directive**
flags	→	`-e` *name*
	→	`-f` *long*
	→	`-ild`
	→	`-l`*name*
	→	`-m`
	→	`-o` *filename*
	→	`-r`
	→	`-s`
	→	`-t`
	→	`-u` *name*
	→	`-x`
	→	`-z`
	→	`-F`
	→	`-L`*pathname*
	→	`-M`
	→	`-N`
	→	`-S`
	→	`-V`
	→	`-VS` *long*
name	→	Any valid symbol name.
long	→	Any valid long integer constant.
filename	→	Any valid A/UX operating system filename; it can include a full or partial pathname.
pathname	→	Any valid A/UX operating system pathname (full or partial).

* These appear to be circular references, but in practice they are (eventually) resolved by definition to a defined element.

16 COFF Reference

This chapter describes the Common Object File Format (COFF). COFF is the output file produced on A/UX systems by the assembler (`as`) and the link editor (`ld`). The term "common" refers to how this format is used on a number of processors and operating systems, including A/UX.

COFF is flexible enough to meet the demands of most jobs, yet simple enough to be easily incorporated into existing projects. Some of the key features of COFF are as follows:

- Applications can add system-dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information that debuggers and other applications use.
- You can make some modifications in the object file construction at compile time.

COFF structure

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains the following elements:

- a file header
- optional header information
- a table of section headers
- data corresponding to the section header
- relocation information
- line numbers
- a symbol table
- a string table

Figure 16-1 shows the overall structure.

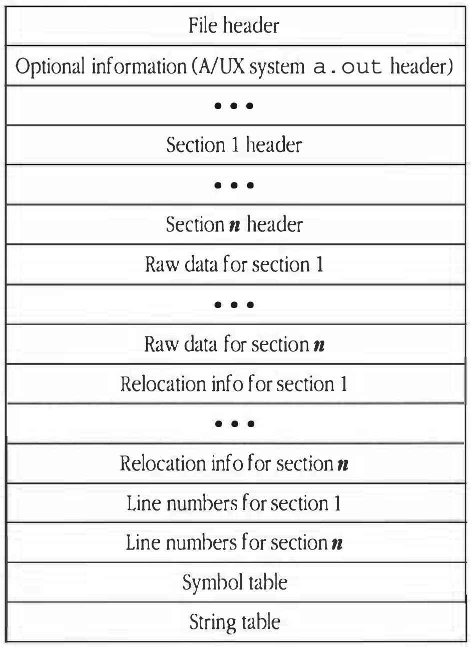
File header
Optional information (A/UX system `a.out` header)
• • •
Section 1 header
• • •
Section *n* header
Raw data for section 1
• • •
Raw data for section *n*
Relocation info for section 1
• • •
Relocation info for section *n*
Line numbers for section 1
Line numbers for section *n*
Symbol table
String table

Figure 16-1 Object file format

The last four sections (relocation, line numbers, symbol table, and string table) might be missing if the program is linked with the `-s` option of the link editor or if the relocation (line number) information, symbol table, and string table are removed by the `strip` command.

The line number information does not appear unless you compile the program with the compiler (`cc`) `-g` option. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters. An object file that contains no errors or unresolved references can be executed.

section	A **section** is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. There are three default sections: `.text`, `.data`, and `.bss`. Additional sections accommodate multiple text or data segments, shared data segments, or user-specified sections. When the file is executed, however, the A/UX operating system loads only the `.text` and `.data` memory. The kernel clears the `.bss` section. Executables using a shared library have additional sections: `.lib` and dummy sections corresponding to the target shared object. An `.init` section specified for a shared-library executable file is placed within a `.text` section of the object file.
physical address	This is the physical location in memory where a section is loaded.
virtual address	This is the offset of a section with respect to the beginning of its segment or region. All relocatable references in a section assume that the section occupies the virtual address at execution time.

File header

The file header contains the 20 bytes of information shown in Table 16-1. The last two bytes are flags used by `ld` and object file utilities. For more explicit information regarding the C language file header structure, see `filehdr`(4) in *A/UX Programmer's Reference*.

Table 16-1 File header contents

Bytes	Declaration	Name	Description
0–1	`unsigned short`	`f_magic`	Magic number as defined by the symbol `MAGIC` in the file `a.out.h`
2–3	`unsigned short`	`f_nscns`	Number of section headers (equals the number of sections)
4–7	`long int`	`f_timdat`	Time and date stamp indicating when the file was created relative to the number of elapsed seconds since 00 00:00 GMT, January 1, 1970
8–11	`long int`	`f_symptr`	File pointer containing the starting address of the symbol table
12–15	`long int`	`f_nsyms`	Number of entries in the symbol table
16–17	`unsigned short`	`f_opthdr`	Number of bytes in the optional header
18–19	`unsigned short`	`f_flags`	Flags

The size of optional header information (`f_opthdr`) is used by all referencing programs that seek to the beginning of the section header table. This enables the same utility programs to work correctly on files originally targeted for different systems. On a VAX system, the optional header is 28 bytes.

Magic numbers

The **magic number** specifies the machine on which the object file is executable. The magic number for A/UX is 0520.

For a complete list of all currently defined magic numbers, refer to the header file `filehdr.h`.

Flags

The last two bytes of the file header are flags that describe the type of the object file. The A/UX version of COFF has no use for some of these, but they are included here for commonality. The currently defined flags are shown in Table 16-2.

Table 16-2 File header flags

Mnemonic	**Flag**	**Meaning**
`F_RELFLG`	`00001`	Relocation information stripped from the file
`F_EXEC`	`00002`	File is executable (that is, no unresolved external references)
`F_LNNO`	`00004`	Line numbers stripped from file
`F_LSYMS`	`00010`	Local symbols stripped from file
`F_MINMAL`	`00020`	Not used by A/UX
`F_UPDATE`	`00040`	Not used by A/UX
`F_SWABD`	`00100`	This file had its bytes **swabbed** (that is, the bytes of symbol table name entries are reversed)
`F_AR16WR`	`00200`	Created on an AR16WR machine (PDP-11)
`F_AR32WR`	`00400`	Created on an AR32WR machine (VAX)
`F_AR32W`	`01000`	Created on an AR32W machine (M68000)
`F_PATCH`	`02000`	Not used by A/UX
`F_NODF`	`02000`	(Minimal file only) No decision functions for replaced functions

Note: AR16WR defines the machine architecture (AR) as 16 bits per word (16), right-to-left byte order with the least significant byte first (WR); AR32WR defines the machine architecture (AR) as 32 bits per word (32), right-to-left byte order with the least significant byte first (WR); and AR32W defines the machine architecture (AR) as 32 bits per word (32), left-to-right byte order with the most significant byte first (W).

File header declaration

The C structure declaration for the file header is given in Figure 16-2. You can find this declaration in the header file `filehdr.h`. See `filehdr`(4) in *A/UX Programmer's Reference*.

Figure 16-2 File header declaration

```
struct filehdr  {
  unsigned short  f_magic;     /* magic number */
  unsigned short  f_nscns;     /* number of sections */
      long        f_timdat;    /* time/date stamp */
      long        f_symptr;    /* file ptr to symtab */
      long        f_nsyms;     /* # symtab entries */
  unsigned short  f_opthdr;    /* sizeof (opt hdr) */
  unsigned short  f_flags;     /* flags */
};
#define FILHDR  struct filehdr
#define FILHSZ  sizeof(FILHDR)
```

Optional header information

The template for optional information varies among the different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that particular operating system uses, without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions) can be made to work properly on any common object file by using the size of optional header information in bytes 16-17 of the file header `f_opthdr`.

Standard A/UX system `a.out` header

By default, files produced by the link editor always have a standard A/UX System `a.out` header in the optional header field. The fields of the optional header are described in Table 16-3.

Table 16-3 Optional header contents

Bytes	Declaration	Name	Description
0–1	`short`	`magic`	Magic number
2–3	`short`	`vstamp`	Version stamp
4–7	`long int`	`tsize`	Size of text in bytes
8–11	`long int`	`dsize`	Size of initialized data in bytes
12–15	`long int`	`bsize`	Size of uninitialized data in bytes
16–19	`long int`	`entry`	Entry point
20–23	`long int`	`text_start`	Base address of text
24–27	`long int`	`data_start`	Base address of data

The magic number in the optional header supplies operating-system-dependent information about the object file, whereas the magic number in the file header specifies the machine on which the object file runs. The magic number in the optional header supplies information telling the operating system of that machine how that file should be executed. The magic numbers recognized by the A/UX operating system are shown in Table 16-4.

The magic number for the A/UX operating system is a machine-dependent constant that can be found in the header file `a.out.h`. See `a.out`(4) in *A/UX Programmer's Reference*.

Table 16-4 A/UX magic numbers

Value	Meaning
0407	The text segment is not write protected or sharable; the data segment is contiguous with the text segment.
0410	The data segment starts at the next segment following the text segment and the text segment is write-protected.
0413	The text segment is demand-paged from the file system, with separate instruction and data space.

Optional header declaration

The C language structure declaration used for the A/UX system `a.out` file header is given in Figure 16-3. This declaration can be found in the header file `aouthdr.h`.

Figure 16-3 `aouthdr` declaration

```
typedef struct aouthdr {
short magic;        /* magic number */
short vstamp;       /* version stamp */
long tsize;         /* text size (bytes) padded */
                    /*  to word boundary */
long dsize;         /* initialized data size */
long bsize;         /* uninitialized data size */
long entry;         /* entry point */
long text_start;    /* base of text, this file */
long data_start     /* base of data, this file */
} AOUTHDR;
```

Section headers

Every object file has a table of section headers to specify the layout of data within the file. Every section in an object file also has its own header. The section header table has one entry for every section in the file. Each entry contains descriptive information about the section, as shown in Table 16-5.

Table 16-5 Section header contents

Bytes	Declaration	Name	Description
0–7	`char`	`s_name`	8-char null padded section name
8–11	`long int`	`s_paddr`	Physical address of section
12–15	`long int`	`s_vaddr`	Virtual address of section
16–19	`long int`	`s_size`	Section size in bytes*
20–23	`long int`	`s_scnptr`	File pointer to raw data†
24–27	`long int`	`s_relptr`	File pointer to relocation entries†
28–31	`long int`	`s_lnnoptr`	File pointer to line number entries†
32–33	`unsigned short`	`s_nreloc`	Number of relocation entries
34–35	`unsigned short`	`s_nlnno`	Number of line number entries
36–39	`long int`	`s_flags`	Flags

* The size of a section is always padded to a multiple of 4 bytes.

† File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the A/UX operating system function `fseek(3S)`.

Flags

The lower four bits of the flag field indicate a section type, as shown in Table 16-6.

Table 16-6 Section header flags

Mnemonic	Flag	Meaning
`STYP_REG`	`0x00`	Regular section (allocated, relocated, loaded).
`STYP_DSECT`	`0x01`	Dummy section (not allocated, relocated, not loaded).
`STYP_NOLOAD`	`0x02`	No-load section (allocated, relocated, not loaded).
`STYP_GROUP`	`0x04`	Grouped section (formed from input sections).
`STYP_PAD`	`0x08`	Padding section (not allocated, not relocated, loaded).
`STYP_COPY`	`0x10`	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally).
`STYP_TEXT`	`0x20`	Section contains executable text only.
`STYP_DATA`	`0x40`	Section contains initialized data only.

Table 16-6 Section header flags *(continued)*

Mnemonic	Flag	Meaning
STYP_BSS	0x80	Section contains only uninitialized data.
STYP_LIB	0x200	Section contains the shared-library pathname (treated similarly to STYP_NOLOAD).
STYPE_INIT	0x400	Section contains shared-library initialization fragments (treated similarly to STYP_TEXT).

Section header declaration

The C structure declaration for the section headers is described in Figure 16-4. You can find this declaration in the header file `scnhdr.h` (see `scnhdr`(4) in *A/UX Programmer's Reference*):

Figure 16-4 Section header declaration

```
struct scnhdr  {
      char     s_name[8];       /* section name */
      long     s_paddr;         /* physical address */
      long     s_vaddr;         /* virtual address */
      long     s_size;          /* section size */
      long     s_scnptr;        /* file pointer to section */
                               /*  raw data */
      long     s_relptr;        /* file pointer to */
                               /*  relocation */
      long     s_lnnoptr;       /* file pointer to line */
                                /*  number */
      unsigned short s_nreloc;  /* # relocation entries */
      unsigned short s_nlnno;   /* # line number entries */
      long s_flags              /* flags */
};
#define SCNHDR  struct scnhdr
#define SCNHSZ  sizeof(SCNHDR)
```

`.bss` section header

The one deviation from the rule in the section header table is the entry for uninitialized data in a `.bss` section. A `.bss` section has a size, symbols that refer to it, and symbols that are defined in it. At the same time, a `.bss` section has no relocation entries, no line number entries, and no data. Therefore, a `.bss` section has an entry in the section header table, but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a `.bss` section header, are zero.

Sections

Section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a full word boundary in the file.

Files produced by the `cc` compiler and the `as` assembler always contain three sections: `.text`, `.data`, and `.bss`. The `.text` section contains the instruction text (that is, executable code); the `.data` section contains initialized data variables; and the `.bss` section contains uninitialized data variables.

The link editor `SECTIONS` directives (see Chapter 15, "`ld` Reference") let you perform the following functions:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections

If you do not include any `SECTIONS` directives, each input section appears in an output section of the same name. For example, if a number of object files from the compiler are linked (each containing the three sections `.text`, `.data`, and `.bss`), the output object file also contains those three sections. Executables using shared libraries have additional sections: `.lib` containing the pathname to shared targets and additional dummy sections (not loaded) corresponding to the sections in the shared target object.

Relocation information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the 10-byte format, as shown in Table 16-7.

Table 16-7 Relocation section contents

Bytes	Declaration	Name	Description
0–3	`long int`	`r_vaddr`	(Virtual) address of reference
4–7	`long int`	`r_symndx`	Symbol table index
8–9	`unsigned short`	`r_type`	Relocation type

The first four bytes of the entry make up the virtual address of the text or data to which the entry applies. The next field is the index, counted from zero, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

The currently recognized relocation types are given in Table 16-8 and are documented in the header file `reloc.h`.

Table 16-8 VAX and M68000 relocation types

Mnemonic	Flag	Meaning
`R_ABS`	0	Reference is absolute; no relocation is necessary; the entry is ignored
`R_RELBYTE`	017	Direct 8-bit reference to the symbol virtual address
`R_RELWORD`	020	Direct 16-bit reference to the symbol virtual address
`R_RELLONG`	021	Direct 32-bit reference to the symbol virtual address (a VAX relocation type)
`R_PCRBYTE`	022	A PC-relative 8-bit reference to the symbol virtual address
`R_PCRWORD`	024	A PC-relative 16-bit reference to the symbol virtual address
`R_PCRLONG`	024	A PC-relative 32-bit reference to the symbol virtual address

On VAX processors, relocation of a symbol index of -1 indicates that the amount by which the section is being relocated is added to the relocatable address. In other words, the relative difference between the current segment start address and the program load address is added to the relocatable address.

The `as` assembler automatically generates relocation entries, which are then used by the link editor to resolve external references in the file.

Relocation entry declaration

The structure declaration for relocation entries is given in Figure 16-5. This declaration can be found in the header file `reloc.h`.

Figure 16-5 Relocation entry declaration

```
struct reloc {
  long r_vaddr;            /* ref virt addr */
  long r_symndx;           /* index into symtab */
  unsigned short r_type;   /* reloc type */
};
#define RELOC struct reloc
#define RELSZ 10           /* sizeof (RELOC) */
```

Line numbers

When invoked with the `-g` option, the A/UX system compilers (`cc`, `f77`) generate an entry in the object file for every C language source line where a break point can be inserted. You can then reference line numbers using a software debugger like `sdb`. All line numbers in a section are grouped by functions, as shown in Figure 16-6.

The first entry in a function grouping has line number zero and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries appear in increasing order of address.

Symbol index	0
Physical address	Line number
Physical address	Line number
• • •	• • •
Symbol index	0
Physical address	Line number
Physical address	Line number

Figure 16-6 Line number grouping

Line number declaration

Figure 16-7 contains the structure declaration currently used for line number entries. This declaration can be found in the header file `linenum.h`.

Figure 16-7 Line number entry declaration

```
strut lineno {
   union {
      long l_symndx;           /* symbol table index of
                               function name */
      long l_paddr;            /* physical address of line
                               number */
   } l_addr;
   unsigned short l_lnno;      /* line number */
};

#define LINENO  struct lineno
#define LINESZ   6             /* sizeof (LINENO) */
```

Symbol table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the symbol table in the sequence shown in Figure 16-8.

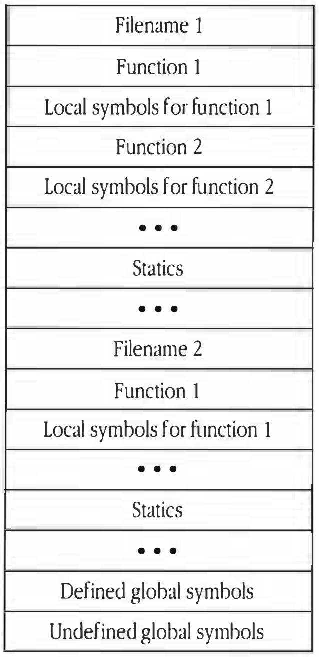

Figure 16-8 COFF global symbol table

The word *statics* means symbols defined in the C language storage class `static` outside any function. The symbol table consists of at least one fixed-length entry per symbol, with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the name (null-padded), structure value, type, and other information.

Special symbols

The symbol table contains some special symbols that are generated by the `cc` compiler, the `as` assembler, and other tools, as listed in Table 16-9.

Table 16-9 Special symbols in the symbol table

Symbol	**Meaning**
`.file`	Filename
`.text`	Address of `.text` section
`.bss`	Address of `.bss` section
`.init`	Address of `.init` section (shared-library routine; contains initialization)
`.lib`	Address of `.lib` section (shared-library routine; contains target pathname)
`.bb`	Address of start of inner block
`.eb`	Address of end of inner block
`.ef`	Address of end of function
`.target`	Pointer to the structure or union returned by a function
`.`*x*`fake`	Dummy tag name for structure, union, or enumeration
`.eos`	End of members of structure, union, or enumeration
`_etext, etext`	Next available address after the end of the output section `.text`
`_edata, edata`	Next available address after the end of the output section `.data`
`_end, end`	Next available address after the end of the output section `.bss`

Six of these special symbols occur in pairs. The `.bb` and `.eb` symbols indicate the boundaries of inner blocks. A `.bf` and `.ef` pair brackets each function and `.`*x*`fake` and `.eos` form a pair that names and defines the limit of structures, unions, and enumerations that were not named. The `.eos` symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the `cc` compiler invents a name to be used in the symbol table. The name chosen for the symbol table is `.`*x*`fake`, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are `.0fake`, `1fake`, and `.2fake`.

Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entry.

Inner blocks

The C language defines a **block** as a compound statement that begins and ends with braces (`{` and `}`). An **inner block** is a block that occurs within a function (which is also a block), such as `if`, `while` or `switch`.

For each inner block that has local symbols defined, a special symbol, `.bb`, is put in the symbol table immediately before the first local symbol of that block. Another special symbol, `.eb`, is put in the symbol table immediately after the last local symbol of that block. Figure 16-9 shows this sequence.

Figure 16-9 Special symbols

`.bb`

Local symbols for that block:

`.eb`

Because inner blocks can be nested by several levels, the `.bb/.eb` pairs and associated symbols also can be nested. The code illustrated in Figure 16-10 is used as an example of nested blocks. The symbol table built for the coding example in Figure 16-10 is shown in Figure 16-11.

Figure 16-10 Nested blocks

```
{                       /* block 1 */
   int i;
   char c;
   {                    /* block 2 */
      long a;
      {                 /* block 3 */
         int x;
      }                 /* block 3 */
   }                    /* block 2 */
         {              /* block 4 */
            long i;
         }              /* block 4 */
}                       /* block 1 */
```

Figure 16-11 Example of the symbol table

`.bb` for block 1

Local symbols for block 1:

`i`

`c`

`.bb` for block 2

Local symbols for block 2:

`a`

`.bb` for block 3

Local symbols for block 3:

`x`

`.eb` for block 3

`.eb` for block 2

`.bb` for block 4

Local symbols for block 4:

`.eb` for block 4

`.eb` for block 1

Symbols and functions

For each function, a special symbol, `.bf`, is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol, `.ef`, is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in the following example:

Function name:

`.bf`

Local symbol:

`.ef`

If the return value of the function is a structure or union, a special symbol, `.target`, is put between the function name and the `.bf`. The sequence is shown here:

Function name:

```
.target
.bf
```

Local symbol:

```
.ef
```

The `cc` compiler invents `.target` to store the function-returned structure or union. The symbol `.target` is an automatic variable with *pointer* type. Its value field in the symbol is always zero.

Symbol table entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain the 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Table 16-10. The declarations can be found in `syms.h` header file.

It should be noted that indexes for symbol table entries begin at zero and count upward. Each auxiliary entry also counts as one symbol.

Table 16-10 Symbol table entry format

Bytes	Declaration	Name	Description
0–7	`char`	`_name`	8-character null-padded symbol name or an offset to a symbol name stored in the string table
8–11	`long int`	`n_value`	Symbol value; storage class dependent
12–13	`short`	`n_scnum`	Section number of symbol
14–15	`unsigned short`	`n_type`	Basic and derived type specification
16	`char`	`n_sclass`	Storage class of symbol
17	`char`	`n_numaux`	Number of auxiliary entries

The first eight bytes in the symbol table entry are the symbol name field. This field is defined as the union of a character array and two `long` types. A symbol name can be up to 50 characters long. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, the entire symbol name is stored in the string table. In this case, the eight bytes contain two `long` integers; the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Because there can be no symbols with a null name, the zeros on the first four bytes serve to distinguish a symbol table entry with an offset from one with a name in the first eight bytes, as shown in Table 16-11.

Table 16-11 Name field

Bytes	Declaration	Name	Description
0–7	`char`	`n_name`	8-character null-padded symbol name
0–3	`long`	`n_zeros`	Zer● in this field indicates the name is in the string table
4–7	`long`	`n_offset`	Offset of the name in the string table

Some special symbols are generated by the compiler and link editor, as discussed in "Special Symbols," earlier in this chapter. Special symbol names always start with a dot, such as `.file`, `.5fake`, and `.bb`.

The storage class field has one of the values described in Table 16-12. You can find these `define` statements in the header file `storclass.h`.

Table 16-12 Storage classes

Mnemonic	Value	Storage class
C_EFCN	-1	Physical end of a function
C_NULL	0	-
C_AUTO	1	Automatic variable
C_EXT	2	External symbol
C_STAT	3	Static
C_REG	4	Register variable
C_EXTDEF	5	External definition

(continued)➡

Table 16-12 Storage classes *(continued)*

Mnemonic	Value	Storage class
`C_LABEL`	6	Label
`C_ULABEL`	7	Undefined label
`C_MOS`	8	Member of structure
`C_ARG`	9	Function argument
`C_STRTAG`	10	Structure tag
`C_MOU`	11	Member of union
`C_UNTAG`	12	Union tag
`C_TPDEF`	13	Type definition
`C_USTATIC`	14	Uninitialized static
`C_ENTAG`	15	Enumeration tag
`C_MOE`	16	Member of enumeration
`C_REGPARM`	17	Register parameter
`C_FIELD`	18	Bit field
`C_BLOCK`	100	Beginning and end of block
`C_FCN`	101	Beginning and end of function
`C_EOS`	102	End of structure
`C_FILE`	103	Filename
`C_LINE`	104	Used only by utility programs
`C_ALIAS`	105	Duplicate tag
`C_HIDDEN`	106	Like `static`, used to avoid name conflicts

All these storage classes, except for `C_ALIAS` and `C_HIDDEN`, are generated by the `cc` compiler or `as` assembler. They are not used by any A/UX system tools.

There are some "dummy" storage classes defined in the header file that are used only internally by the C compiler (`cc`) and the assembler (`as`). These storage classes are as follows:

`C_EFCN`

`C_EXTDEF`

`C_ULABEL`

`C_USTATIC`

`C_LINE`

Some special symbols are restricted to certain storage classes, listed in Table 16-13.

Table 16-13 Storage class by special symbols

Special symbol	Storage class
`.file`	`C_FILE`
`.bb`	`C_BLOCK`
`.eb`	`C_BLOCK`
`.bf`	`C_FCN`
`.ef`	`C_FCN`
`.target`	`C_AUTO`
`.`*x*`fake`	`CSTRTAG, C_UNTAG, C_ENTAG`
`.eos`	`C_EOS`
`.text`	`C_STAT`
`.data`	`C_STAT`
`.bss`	`C_STAT`

Some storage classes are used only for certain special symbols as shown in Table 16-14.

Table 16-14 Restricted storage class

Storage class	Special symbol
`C_BLOCK`	`.bb, .eb`
`C_FCN`	`.bf, .ef`
`C_EOS`	`.eos`
`C_FILE`	`.file`

The meaning of a symbol value depends on its storage class. This relationship is summarized in Tables 16-15 and 16-16.

If a symbol is the last symbol in the object file and has storage class `C_FILE` (`.file` symbol), its value equals the symbol table entry index of the first global symbol. That is, the `.file` entries form a one-way linked list in the symbol table. If there are no more `.file` entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to their virtual address. When the section is relocated by the link editor, the value of these symbols changes.

Table 16-15 Storage class and value

Storage class	Meaning
`C_AUTO`	Stack offset in bytes
`C_EXT`	Relocatable address
`C_STAT`	Relocatable address
`C_REG`	Register number
`C_LABEL`	Relocatable address
`C_MOS`	Offset in bytes
`C_ARG`	Stack offset in bytes
`C_STRTAG`	0
`C_MOU`	Offset
`C_UNTAG`	0
`C_TPDEF`	0
`C_ENTAG`	0
`C_MOE`	Enumeration value
`C_REGPARM`	Register number
`C_FIELD`	Bit displacement
`C_BLOCK`	Relocatable address
`C_FCN`	Relocatable address
`C_EOS`	Size
`C_EOS`	Size
`C_FILE`	(See text)

Section numbers are declared in the header file `syms.h` and are listed in Table 16-16.

Table 16-16 Section number

Mnemonic	Section number	Meaning
`N_DEBUG`	-2	Special symbolic debugging symbol
`N_ABS`	-1	Absolute symbol
`N_UNDEF`	0	Undefined external symbol
`N_SCNUM`	1–077767	Section number where symbol was defined

A special section number (-2) marks symbolic debugging symbols, including structure (or union or enumeration) tag names, `typedef` statements, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and `.eos` symbols. The `.text`, `.data`, and `.bss` symbols default to section numbers 1, 2, and 3, respectively.

With one exception, a section number of zero indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply defined external symbol (for example, a Fortran COMMON directive or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is zero and the value of the symbol is a positive number giving the size of the symbol. When the files are combined, the link editor combines all the input symbols into one symbol with the section number of the `.bss` section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol, and the value becomes the address of the symbol. This is the only case where a symbol has a section number of zero and a nonzero value.

Symbols having certain storage classes are also restricted to certain section numbers. They are shown in Table 16-17.

Table 16-17 Section number and storage class

Storage class	**Section number**
`C_AUTO`	`N_ABS`
`C_EXT`	`N_ABS, N_UNDEF, N_SCNUM`
`C_STAT`	`N_SCNUM`
`C_REG`	`N_ABS`
`C_LABEL`	`N_UNDEF, N_SCNUM`
`C_MOS`	`N_ABS`
`C_ARG`	`N_ABS`
`C_STRTAG`	`N_DEBUG`
`C_MOU`	`N_ABS`
`C_UNTAG`	`N_DEBUG`
`C_TPDEF`	`N_DEBUG`
`C_ENTAG`	`N_DEBUG`
`C_MOE`	`N_ABS`
`C_REGPARM`	`N_ABS`
`C_FIELD`	`N_ABS`
`C_BLOCK`	`N_SCNUM`
`C_FCN`	`N_SCNUM`
`C_EOS`	`N_ABS`
`C_FILE`	`N_DEBUG`
`C_ALIAS`	`N_DEBUG`

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by `cc`. The VAX and MC68000-family `cc` compilers generate this information only if the `-g` option is used. Each symbol has exactly one basic or fundamental type, but can have more than one derived type. The format of the 16-bit type entry is

d6	d5	d4	d3	d2	d1	typ

Bits 0 through 3, called `typ`, indicate one of the fundamental types shown in Table 16-18.

Table 16-18 Fundamental types

Mnemonic	Value	Type
T_NULL	0	Type not assigned
T_ARG	1	Function argument (used only by compiler)
T_CHAR	2	Character
T_SHORT	3	Short integer
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Floating point
T_DOUBLE	7	Double word
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_MOE	11	Member of enumeration
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned long

Bits 4 through 15 are arranged as six 2-bit fields marked `d1` through `d6`. These `d` fields represent levels of the derived types shown in Table 16-19.

Table 16-19 Derived types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

The following examples demonstrate interpretation of the symbol table entry representing type:

`char *`*`func`*`();`

Here, *func* is the name of a function that returns a pointer to a character. The fundamental type of *func* is 2 (character), the `d1` field is 2 (function), and the `d2` field is 1 (pointer). Therefore, the type word in the symbol table for *func* contains the hexadecimal number 0x62, which is interpreted to mean "a function that returns a pointer to a character."

`short *`*`tabptr`*`[10][25][3];`

Here, *tabptr* is a three-dimensional array of pointers to short integers. The fundamental type of *tabptr* is 3 (short integer); each of the `d1`, `d2`, and `d3` fields contains a 3 (array), and the `d4` field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f7, indicating "a three-dimensional array of pointers to short integers."

Table 16-20 shows the type entries that are legal for each storage class.

Conditions for the `d` entries apply to `d1` through `d6`, except that it is impossible to have two consecutive derived types of *function.*

Although *function* arguments can be declared as *arrays,* they are changed to *pointers* by default. Therefore, no *function* argument can have *array* as its first derived type.

Table 16-20 Type entries by storage class

	`d` entry			
Storage class	**Function**	**Array**	**Pointer**	`typ` **entry basic type**
C_AUTO		×	×	Any except T_MOE
C_EXT	×	×	×	Any except T_MOE
C_STAT	×	×	×	Any except T_MOE
C_REG			×	Any except T_MOE
C_LABEL				T_NULL
C_MOS		×	×	Any except T_MOE
C_ARG	×		×	Any except T_MOE
C_STRTAG				T_STRUCT
C_MOU		×	×	Any except T_MOE
C_UNTAG				T_UNION
C_TPDEF		×	×	Any except T_MOE
C_ENTAG				T_ENUM
C_MOE				T_MOE
C_REGPARM			×	Any except T_MOE
C_FIELD				T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK				T_NULL
C_FCN				T_NULL
C_EOS				T_NULL
C_FILE				T_NULL
C_ALIAS				T_STRUCT, T_UNION, T_ENUM

The C language structure declaration for the symbol table entry is given in Figure 16-12. This declaration can be found in the header file `syms.h`.

Figure 16-12 Symbol table entry declaration

```
struct syment {
   union {
      char _n_name[SYMNMLEN]; /* symbol name*/
      struct {
         long _n_zeros;    /* symbol name */
         long _n_offset;    /* location in string table */
      } _n_n;
      char *_n_nptr[2];     /* allows overlaying */
      } _n;
         long     n_value;  /* symbol value */
         short    n_scnum;  /* section number */
   unsigned short n_type;   /* type & derived */
         char     n_sclass; /* storage class */
         char     n_numaux; /* # of aux entries */
};
#define n_name     _n._n_name
#define n_nptr     _n._n_nptr[1]
#define n_zeros   _n._n_n._n_zeros
#define n_offset   _n._n_n._n_offset
#define SYMNMLEN   8
#define SYMENT     struct syment
#define SYMESZ     18     /* symbol table entry size */
```

Auxiliary table entries

Currently, there is, at most, one auxiliary entry per symbol. The auxiliary table entry contains the same number of bytes as the symbol table entry. Unlike symbol table entries, however, the format of an auxiliary table entry of a symbol depends on its type and storage class. Table 16-21 lists auxiliary table entry formats by type and storage class.

Table 16-21 Auxiliary symbol table entries

		Type entry		
Name	**Storage class**	d2	typ	**Auxiliary entry format**
.file	C_FILE	DT_NON	T_NULL	Filename
.text, .data, .bss	C_STAT	DT_NON	T_NULL	Section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	Tag name
.eos	C_EOS	DT_NON	T_NULL	End of structure
fcname	C_EXT C_STAT	DT_FCN	Any except T_MOE	Function
arrname	C_AUTO C_STAT C_MOS C_MOU C_TPDEF	DT_ARY	Any except T_MOE	Array
.bb	C_BLOCK	DT_NON	T_NULL	Beginning of block
.eb	C_BLOCK	DT_NON	T_NULL	End of block
.bf, .ef	C_FCN	DT_NON	T_NULL	Beginning and end of function
Name related to structure, union, enumeration	C_AUTO C_STAT C_MOS C_MOU C_TPDEF	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION, T_ENUM	Name related to structure, union, enumeration

In the preceding table, *tagname* means any symbol name, including the special symbol .xfake, and *fcname* and *arrname* represent any symbol name.

Any symbol that satisfies more than one condition should have a union format in its auxiliary entry. Symbols that do not satisfy any of the above conditions should not have any auxiliary entry.

Each of the auxiliary table entries for a filename contains a 14-character filename in bytes zero through 13. The remaining bytes are zero, regardless of the size of the entry.

The auxiliary table entries for sections have the format as shown in Table 16-22.

Table 16-22 Format for sections in auxiliary table

Bytes	Declaration	Name	Description
0–3	`long int`	`x_scnlen`	Section length
4–6	`unsigned short`	`x_nreloc`	Number of relocation entries
6–7	`unsigned short`	`x_nlinno`	Number of line numbers
8–17	–	dummy	Unused (filled with zeros)

The auxiliary table entries for tag names have the format shown in Table 16-23. The auxiliary table entries for the end of structures have the format shown in Table 16-24. The auxiliary table entries for functions have the format shown in Table 16-25. The auxiliary table entries for arrays have the format shown in Table 16-26. The auxiliary table entries for the beginning of blocks have the format shown in Table 16-27. The auxiliary table entries for the end of blocks have the format shown in Table 16-28. The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Table 16-29.

Table 16-23 Format for tag names

Bytes	Declaration	Name	Description
0–5	–	dummy	Unused (filled with zeros)
6–7	`unsigned short`	`x_size`	Size of `struct`, `union`, and enumeration
8–11	–	dummy	Unused (filled with zeros)
12–15	`long int`	`x_endndx`	Index of next entry beyond this structure, union, or enumeration
16–17	–	dummy	Unused (filled with zeros)

Table 16-24 Format for end of structures

Bytes	Declaration	Name	Description
0–3	`long int`	`x_tagndx`	Tag index
4–5	–	dummy	Unused (filled with zeros)
6–7	`unsigned short`	`x_size`	Size of `struct`, `union`, or enumeration
8–17	–	dummy	Unused (filled with zeros)

Table 16-25 Format for functions

Bytes	Declaration	Name	Description
0–3	`long int`	`x_tagndx`	Tag index
4–7	`long int`	`x_fsize`	Size of function (in bytes)
8–11	`long int`	`x_lnnoptr`	File pointer to line number
12–15	`long int`	`x_endndx`	Index of next entry beyond this function
16–17	`unsigned short`	`x_tvndx`	Index of the function address in the transfer vector table (not used by A/UX operating system)

Table 16-26 Format for arrays

Bytes	Declaration	Name	Description
0–3	`long int`	`x_tagndx`	Tag index
4–5	`unsigned short`	`x_lnno`	Line number of declaration
6–7	`unsigned short`	`x_size`	Size of array
8–9	`unsigned short`	`x_dimen[0]`	First dimension
10–11	`unsigned short`	`x_dimen[1]`	Second dimension
12–13	`unsigned short`	`x_dimen[2]`	Third dimension
14–15	`unsigned short`	`x_dimen[3]`	Fourth dimension
16–17	–	dummy	Unused (filled with zeros)

Table 16-27 Format for beginning of block

Bytes	Declaration	Name	Description
0–3	–	dummy	Unused (filled with zeros)
4–5	`unsigned short`	`x_lnno`	C-source line number
6–11	–	dummy	Unused (filled with zeros)
12–15	`long int`	`x_endndx`	Index of next entry past this block
16–17	–	dummy	Unused (filled with zeros)

Table 16-28 Format for end of block

Bytes	Declaration	Name	Description
0–3	–	dummy	Used (filled with zeros)
4–5	`unsigned short`	`x_lnno`	C-source line number
6–17	–	dummy	Unused (filled with zeros)

Table 16-29 Format for structures, unions, and enumerations

Bytes	Declaration	Name	Description
0–3	`long int`	`x_tagndx`	Tag index
4–5	–	dummy	Unused (filled with zeros)
6–7	`unsigned short`	`x_size`	Size of the `structure`, `union`, or enumeration
8–17	–	dummy	Unused (filled with zeros)

Names defined by `typedef` might or might not have auxiliary table entries. For example,

```
typedef struct people STUDENT;

struct people {
   char name[20];
   long id;
   };

typedef struct people EMPLOYEE;
```

The symbol `EMPLOYEE` has an auxiliary table entry in the symbol table, but the symbol `STUDENT` does not.

The C language structure declaration for an auxiliary symbol table entry is given in Figure 16-13. This declaration can be found in the header file `syms.h`.

Figure 16-13 Auxiliary symbol table entry

```
union auxent {
   struct {
      long x_tagndx;
      union {
         struct {
            unsigned short x_lnno;
            unsigned short x_size;
         } x_lnsz;
         long x_fsize;
      } x_misc;
      union {
         struct {
            long x_lnnoptr;
            long x_endndx;
         } x_fcn;
         struct {
            unsigned short x_dimen[DIMNUM];
         } x_ary;
      } x_fcnary;
         unsigned short x_tvndx;
} x_sym;
   struct {
      char x_fname[FILNMLEN];
   } x_file;
   struct {
      long                            x_scnlen;
```

(continued)➡

```
        unsigned short x_nreloc;
        unsigned short x_nlinno;
    } x_scn;
    struct {
        long                          x_tvfill;
        unsigned short x_tvlen;
        unsigned short x_tvran[2];
    } x_tv;
}
#define FILNMLEN  14
#define DIMNUM     4
#define AUXENT    union auxent
#define AUXESZ    18
```

String table

Symbol table names longer than eight characters are stored contiguously in the string table, with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table are therefore greater than or equal to four.

For example, given a file containing two symbols with names longer than eight characters, `long_name_1` and `another_one`, the string table has the format shown in Table 16-30.

Table 16-30 String table

`'l'`	`'o'`	`'n'`	`'g'`
`'_'`	`'n'`	`'a'`	`'m'`
`'e'`	`'_'`	`'1'`	`'\0'`
`'a'`	`'n'`	`'o'`	`'t'`
`'h'`	`'e'`	`'r'`	`'_'`
`'o'`	`'n'`	`'e'`	`'\0'`

◆ **Note** The index of `long_name_1` in the string table is 4, and the index of `another_one` is 16. ◆

Access routines

Supplied with every standard A/UX system release is a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file. In this way, you can concern yourself with the section you are interested in without knowing all the object file details.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. functions that return the symbol table index for a particular symbol

These routines can be found in the library `libld.a` and are listed, along with a summary of what is available, in *A/UX Programmer's Reference* under `ldfcn`(3X).

Appendix A: Additional Reading

Bach, Maurice J. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1986. (internal algorithms and data structures)

Harbison, Samuel P., and Steele, Guy L., Jr. *C: A Reference Manual.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

Kernighan, Brian W., and Plauger, P. J. *The Elements of Programming Style.* New York: McGraw-Hill, 1974. (coding and design techniques)

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

MC68020 32-Bit Microprocessor User's Manual. 2d ed. Motorola, 1985.

MC68881 Floating-Point Coprocessor User's Manual. Motorola, 1985.

Rochkind, Marc J. *Advanced UNIX Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1985. (UNIX system calls)

System V Interface Definition. AT&T, 1986.

Appendix B: A/UX POSIX Environment

The Institute of Electrical and Electronics Engineers (IEEE) standard Portable Operating System Interface for Computer Environments (POSIX) 1003.1-1990 is a standard developed to promote portability of applications across operating system environments. For more detailed information about A/UX POSIX compliance, see Appendix C, "The A/UX Guide to POSIX." The A/UX POSIX environment is compliant with this standard and with the United States Federal Information Processing Standard (FIPS) #151-1.

This appendix describes the A/UX POSIX environment. There is a new library, `libposix.a`, containing new and modified system calls and subroutines for the A/UX POSIX environment. There are new symbolic constants in several header files and a few new header files. Correct use of POSIX functionality requires programs to be compiled with a new option to `cc`. This appendix provides information on these additions to A/UX and gives some examples of how to use the new functions.

Compiling programs

To compile a program for the POSIX environment, use the `cc` command with the `-ZP` option. For example, to compile file `foo.c` the following command is used:

```
cc -o foo -ZP foo.c
```

The `-ZP` flag ensures that `libposix` is searched before `libc`, links the program with a library module that calls `setcompat`(2) with the `COMPAT_POSIX` flag set, and defines the `_POSIX_SOURCE` feature test macro.

POSIX optional facilities

POSIX specifies numerous optional facilities. These options are indicated by flags defined in the header file `<unistd.h>`. A/UX POSIX supports the following options:

Define name	*Description*
`_POSIX_JOB_CONTROL`	Job control based on the 4.2BSD model is present.
`_POSIX_CHOWN_RESTRICTED`	`chown`(2) can only be called by processes with effective user ID of zero.
`NGROUPS_MAX`	Process permissions include supplementary groups' IDs.
`_POSIX_SAVED_IDS`	The effective user and group IDs are saved by `exec`(2).
`_POSIX_VDISABLE`	Terminal special characters defined in the `c_cc` array can be individually disabled using the value specified by `_POSIX_V_DISABLE`.
`_POSIX_NO_TRUNC`	Pathname components longer than `NAME_MAX` generate an error.

Process compatibility flag

A/UX has a process compatibility flag that is associated with each process. The system calls `setcompat`(2) and `getcompat`(2) are used to change and examine this flag. Where there is conflicting functionality defined by System V and BSD, the process compatibility flag allows applications to select which functionality to use. This flag is also used to support incompatible features defined by POSIX.

The following POSIX options are supported by the corresponding compatibility flags:

POSIX option	*Flag*
`_POSIX_CHOWN RESTRICTED`	`COMPAT_BSDCHOWN`
`_POSIX_NOTRUNC`	`COMPAT_BSDNOTRUNC`

`COMPAT_POSIX` is a composite flag equivalent to all of the following:

```
COMPAT_BSDGROUPS
COMPAT_BSDCHOWN
COMPAT_BSDSIGNALS
COMPAT_BSDTTY
COMPAT_BSDNOTRUNC
COMPAT_EXEC
COMPAT_SETUGID
COMPAT_POSIXFUS
```

New system calls

There are four new system calls in A/UX POSIX. These functions are discussed in more detail in section 2 of the *A/UX Programmer's Reference.*

Function	*Description*
`setpgid`(2)	set process group ID for job control
`sigpending`(2)	examine pending signals
`setsid`(2)	create session and set process group ID
`waitpid`(2)	obtain status information regarding child processes

The POSIX library

All of the functions listed in this section are discussed in more detail in the *A/UX Programmer's Reference*.

Terminal interface control

POSIX specifies a new general terminal interface; this is discussed in `termios`(7P) in *A/UX System Administrator's Reference*. The following functions replace the traditional `ioctl`(2) interface for terminal control. The `tcsetpgrp`() and `tcgetpgrp`() functions are part of the POSIX job control option.

Function	*Reference*	*Description*
`tcdrain`	`tcdrain`(3P)	wait until all written data is transmitted
`tcflow`	`tcdrain`(3P)	suspend or restart input or output
`tcflush`	`tcdrain`(3P)	discard data not transmitted
`tcgetattr`	`tcgetattr`(3P)	get terminal attributes
`tcgetpgrp`	`tcgetpgrp`(3P)	get distinguished process group ID
`tcsendbreak`	`tcdrain`(3P)	send a break
`tcsetattr`	`tcgetattr`(3P)	set terminal attributes
`tcsetpgrp`	`tcsetpgrp`(3P)	set distinguished process group ID

The following functions allow changes to the baud rate in the control structure. A/UX does not support different values for the input and output baud rate; both `cfsetispeed`() and `cfsetospeed`() change the input and output baud rates.

Function	*Reference*	*Description*
`cfgetispeed`	`cfgetospeed`(3P)	return input baud rate
`cfgetospeed`	`cfgetospeed`(3P)	return output baud rate
`cfgetospeed`	`cfgetospeed`(3P)	set input baud rate
`cfsetispeed`	`cfgetospeed`(3P)	set output baud rate

Signals

POSIX specifies new signal functions that are modeled on 4.2BSD signals. Some of the new functions have corollaries in the BSD signal environment:

POSIX function	*BSD function*
`sigaction`(3P)	`sigvec`(2)
`sigprocmask`(3P)	`sigsetmask`(2)
`sigsuspend`(3P)	`sigpause`(2)

There are five functions provided for manipulating signal sets. These routines provide functionality similar to that of the `sigmask( )` macro.

Function	*Reference*	*Description*
`sigaction`	`sigaction`(3P)	examine and change signal action
`sigaddset`	`sigsetops`(3P)	add a signal to a signal set
`sigdelset`	`sigsetops`(3P)	delete a signal from a signal set
`sigfillset`	`sigsetops`(3P)	initialize a signal set to include all POSIX-defined signals
`sigemptyset`	`sigsetops`(3P)	initialize a signal set to exclude all POSIX-defined signals
`sigismember`	`sigsetops`(3P)	determine whether a signal is a member of a signal set
`sigprocmask`	`sigprocmask`(3P)	examine and change blocked signals
`sigsuspend`	`sigsuspend`(3P)	wait for a signal

Configurable system variables

POSIX introduced the following new routines, which allow an application to query environment and system variables at runtime:

Function	*Reference*	*Description*
`fpathconf`	`pathconf`(3P)	get current values of configurable file-related variables
`pathconf`	`pathconf`(3P)	get current values of configurable file-related variables
`sysconf`	`sysconf`(3P)	get values of configurable system variables

Miscellaneous

The POSIX environment has the following new routine:

Function	*Reference*	*Description*
`mkfifo`	`mkfifo`(3P)	make a FIFO special file

Header files and feature test macros

POSIX specifies certain symbols that are defined in header files. The available header files are as follows:

```
unistd.h

sys/types.h

sys/stat.h

fcntl.h

limits.h

utime.h
```

Some of these header files also can define symbols in addition to those defined by POSIX, potentially conflicting with symbols defined by an application program. These potential problems can be dealt with by using feature test macros, which control the visibility of these symbols in the header files required by POSIX.

The rest of this section describes feature test macros and lists the contents of available header files.

Feature test macros

A/UX defines the following feature test macros:

`_SYSV_SOURCE`

`_BSD_SOURCE`

`_AUX_SOURCE`

`_FIPS_151_SOURCE`

The feature test macros `_SYSV_SOURCE` and `_BSD_SOURCE` represent the historical implementations on which A/UX is based. `_AUX_SOURCE` represents extensions to the historical implementations that are specific to A/UX.

The feature test macro `_FIPS_151_SOURCE` represents functionality specific to the initial version of the POSIX FIPS and is present for backward compatibility only. Application programs should not use this feature test macro.

Feature test macros can be invoked on the `cc` command line. See Chapter 2, "`cc` Command Syntax," for a description of the command-line arguments and their effects.

`<unistd.h>`

```
#ifndef __unistd_h
#define __unistd_h

#ifdef _SYSV_SOURCE
/* lockf(..., function, ...) values */
#define F_ULOCK  0       /* Unlock a previously locked */
                         /* region */
```

```
#define F_LOCK   1        /* Lock a region for exclusive */
                          /* use */
#define F_TLOCK  2        /* Test and lock a region for */
                          /* exclusive use */
#define F_TEST   3        /* Test a region for other */
                          /* processes locks */
#endif /* _SYSV_SOURCE */
#ifdef _POSIX_SOURCE
#ifndef NULL
#define     NULL   0
#endif

/* access(..., mode) values */
#define  R_OK  4  /* read permission */
#define  W_OK  2  /* write permission */
#define  X_OK  1  /* execute or search permission */
#define  F_OK  0  /* existence only */

/* lseek(..., whence) values */
#define  SEEK_SET  0  /* beginning of file */
#define  SEEK_CUR  1  /* current position */
#define  SEEK_END  2  /* end of file */

 /* initial file descriptor values */
#define     STDIN_FILENO  0
#define     STDOUT_FILENO 1
#define     STDERR_FILENO 2

 /* POSIX option flags */
#define     _POSIX_JOB_CONTROL       1
#define     _POSIX_SAVED_IDS         1
#define     _POSIX_VERSION           198808L
#define     _POSIX_CHOWN_RESTRICTED  1
#define     _POSIX_NO_TRUNC          1
#define     _POSIX_VDISABLE          0377
```

(continued)➡

```
/* sysconf() names */
#define     _SC_ARG_MAX                 0x00000001
#define     _SC_CHILD_MAX               0x00000002
#define     _SC_CLK_TCK                 0x00000004
#define     _SC_NGROUPS_MAX             0x00000008
#define     _SC_OPEN_MAX                0x00000010
#define     _SC_JOB_CONTROL             0x00000200
#define     _SC_SAVED_IDS               0x00002000
#define     _SC_VERSION                 0x00004000

/* pathconf() names */
#define     _PC_LINK_MAX                0x00020000
#define     _PC_MAX_CANON               0x00040000
#define     _PC_MAX_INPUT               0x00080000
#define     _PC_NAME_MAX                0x00100000
#define     _PC_PATH_MAX                0x00200000
#define     _PC_PIPE_BUF                0x00800000
#define     _PC_CHOWN_RESTRICTED        0x01000000
#define     _PC_NO_TRUNC                0x20000000
#define     _PC_VDISABLE                0x80000000

/* POSIX miscellaneous function declarations */ extern char
*getcwd(), *getlogin(), *ttyname();

#ifdef _FIPS_151_SOURCE
/* POSIX option flags (artifacts from Draft 12) */
#define  _POSIX_CHOWN_SUP_GRP  1
#define  _POSIX_UTIME_OWNER    1
#define  _POSIX_GROUP_PARENT   1
#define  _POSIX_KILL_SAVED     1
#define  _POSIX_EXIT_SIGHUP    1
#define  _POSIX_KILL_PID_NEG1  1
#define  _POSIX_DIR_DOTS       1
#define  _POSIX_PGID_CLEAR     1
#define  _POSIX_V_DISABLE      _POSIX_VDISABLE
```

```
/* sysconf() names (artifacts from Draft 12) */
#define     _SC_PASS_MAX            0x00000020
#define     _SC_PID_MAX             0x00000040
#define     _SC_UID_MAX             0x00000080
#define     _SC_EXIT_SIGHUP         0x00000100
#define     _SC_KILL_PID_NEG1       0x00000400
#define     _SC_KILL_SAVED          0x00000800
#define     _SC_PGID_CLEAR          0x00001000

 /* pathconf() names (artifacts from Draft 12) */
#define     _PC_FCHR_MAX            0x00010000
#define     _PC_PIPE_MAX            0x00400000
#define     _PC_CHOWN_SUP_GRP       0x02000000
#define     _PC_DIR_DOTS            0x04000000
#define     _PC_GROUP_PARENT        0x08000000
#define     _PC_LINK_DIR            0x10000000
#define     _PC_UTIME_OWNER         0x40000000
#define     _PC_V_DISABLE            _PC_VDISABLE
#endif /*                        _FIPS_151_SOURCE */
#endif /*                        _POSIX_SOURCE */
#endif                           /* !__unistd_h */
```

<sys/types.h>

The following types are defined in `<sys/types.h>` for the POSIX environment:

```
#ifndef __sys_types_h
#define __sys_types_h
#define _TYPES_     /* for backwards compatibility */
/*
 * System-dependent parameters and types
 */
typedef char *           caddr_t;
typedef long             clock_t;
typedef short            cnt_t;
typedef long             daddr_t;
typedef unsigned short   dev_t;
typedef int              gid_t;
#ifdef _POSIX_SOURCE
typedef unsigned long    ino_t;
#else
typedef unsigned short   ino_t;
#endif /* _POSIX_SOURCE */
typedef long             key_t;
typedef int              label_t[13];
typedef unsigned short   mode_t;
typedef short            nlink_t;
typedef long             off_t;
typedef long             paddr_t;
typedef int              pid_t;
typedef int              ptrdiff_t;
typedef int              size_t;
typedef long             time_t;
typedef long             ubadr_t; /* physical unibus address
*/
typedef unsigned char    uchar_t;
typedef unsigned short  ushort_t;
typedef int             uid_t;
typedef unsigned int    uint_t;
typedef unsigned long   ulong_t;
typedef unsigned int    wchar_t;
#ifndef NULL
#define NULL            0
#endif /* NULL */
/*
 * To be excluded from visibility control,
 * types must end in _t.
 */
#ifdef _SYSV_SOURCE
typedef     unsigned int   uint;
typedef     unsigned long  ulong;
typedef unsigned char      unchar;
typedef     unsigned short ushort;
#endif /* _SYSV_SOURCE */

#ifdef _BSD_SOURCE
typedef struct fd_set { long fds_bits[1]; } fd_set;
typedef     struct{int r[1];} *physadr;
typedef struct             _quad { long val[2]; } quad;
typedef unsigned char      u_char;
typedef     unsigned short u_short;
typedef unsigned int       u_int;
typedef unsigned long      u_long;
#endif /* _BSD_SOURCE */

#ifdef _AUX_SOURCE
typedef     unsigned long  ino_tl;
#endif /* _AUX_SOURCE */
#endif /* !__sys_types_h */
```

<sys/stat.h>

The header file <sys/stat.h> has the following defines for the POSIX environment:

```
#ifndef __sys_stat_h
#define __sys_stat_h
/*
 * Structure of the result of stat
 */
#ifdef _POSIX_SOURCE
struct stat
{
            dev_t         st_dev;
            short         st_spare0;
            mode_t        st_mode;
            nlink_t       st_nlink;
            int           st_spare1;
            dev_t         st_rdev;
            off_t         st_size;
            time_t        st_atime;
            ino_t         st_ino;
#ifdef _AUX_SOURCE
#define  st_inol  st_ino  /* backward compatibility */
#endif /* _AUX_SOURCE */
            time_t        st_mtime;
            int           st_spare2;
            time_t        st_ctime;
            int           st_spare3;
            long          st_blksize;
            long          st_blocks;
            uid_t         st_uid;
            gid_t         st_gid;
};
```

```
#else /* !_POSIX_SOURCE */
struct      stat
{
            dev_t          st_dev;
            ino_t          st_ino;
            mode_t         st_mode;
            nlink_t        st_nlink;
            short          st_uid;
            short          st_gid;
            dev_t          st_rdev;
            off_t          st_size;
            time_t         st_atime;
#ifdef _AUX_SOURCE
            ino_tl         st_inol;
#else /* !_AUX_SOURCE */
            ulong_t        st_inol;
#endif /* _AUX_SOURCE */
            time_t         st_mtime;
            int            st_spare2;
            time_t         st_ctime;
            int            st_spare3;
            long           st_blksize;
            long           st_blocks;
            long           st_spare4[2];
};
#endif /* _POSIX_SOURCE */

#define    S_ISUID   04000   /* set user id on execution */
#define    S_ISGID   02000   /* set group id on execution */
```

(continued)➡

```
#if defined(_SYSV_SOURCE) || defined(_BSD_SOURCE)
/* historical file type constants */
#define    S_IFMT     0170000  /* type of file */
#define    S_IFIFO    0010000  /* fifo */
#define    S_IFCHR    0020000  /* character special */
#define    S_IFDIR    0040000  /* directory */
#define    S_IFBLK    0060000  /* block special */
#define    S_IFREG    0100000  /* regular */

/* additional (historical) file modes */
#define S_ISVTX  01000 /* save swapped text even after use */
#define S_IREA   00400 /* read permission, owner */
#define S_IWRITE 00200 /* write permission, owner */
#define S_IEXEC  00100 /* execute/search permission, owner */

#define  S_ISFIFO(m)    (((m) & S_IFMT) == S_IFIFO)
#define  S_ISCHR(m)     (((m) & S_IFMT) == S_IFCHR)
#define  S_ISDIR(m)     (((m) & S_IFMT) == S_IFDIR)
#define  S_ISBLK(m)     (((m) & S_IFMT) == S_IFBLK)
#define  S_ISREG(m)     (((m) & S_IFMT) == S_IFREG)
#else
#ifdef _POSIX_SOURCE
/*
 * POSIX doesn't require the historical versions
 * of these file type constants (see above).
 */
#define    _S_IFMT   0170000      /* type of file */
#define    _S_IFIFO  0010000  /* fifo */
#define    _S_IFCHR  0020000  /* character special */
#define    _S_IFDIR  0040000  /* directory */
#define    _S_IFBLK  0060000  /* block special */
#define    _S_IFREG  0100000  /* regular */
```

```
#define     S_ISFIFO(m) (((m) & _S_IFMT) == _S_IFIFO)
#define     S_ISCHR(m)  (((m)  & _S_IFMT) == _S_IFCHR)
#define     S_ISDIR(m)  (((m)  & _S_IFMT) == _S_IFDIR)
#define     S_ISBLK(m)  (((m)  & _S_IFMT) == _S_IFBLK)
#define     S_ISREG(m)  (((m)  & _S_IFMT) == _S_IFREG)
#endif /* _POSIX_SOURCE */
#endif /* _SYSV_SOURCE || _BSD_SOURCE */

#ifdef _BSD_SOURCE
/* additional file types */
#define    S_IFLNK    0120000   /* symbolic link */
#define    S_IFSOCK   0140000  /* socket */

#define     S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)
#define     S_ISSOCK(m)     (((m) & S_IFMT) == S_IFSOCK)
#endif /* _BSD_SOURCE */

#if defined(_SYSV_SOURCE) || defined(_POSIX_SOURCE)
#define S_IRUSR  00400 /* read permission, owner */
#define S_IWUSR  00200 /* write permission, owner */
#define S_IXUSR  00100 /* execute/search permission, owner
*/
#define S_IRWXU   (S_IRUSR | S_IWUSR | S_IXUSR)

#define S_IRGRP  00040 /* read permission, group */
#define S_IWGRP  00020 /* write permission, group */
#define S_IXGRP  00010 /* execute/search permission, group
*/
#define  S_IRWXG   (S_IRGRP | S_IWGRP | S_IXGRP)

#define S_IROTH  00004 /* read permission, other */
#define S_IWOTH  00002 /* write permission, other */
#define S_IXOTH  00001 /* execute/search permission, other
*/
#define  S_IRWXO   (S_IROTH | S_IWOTH | S_IXOTH)
#endif /* _SYSV_SOURCE || _POSIX_SOURCE */
#endif /* !__sys_stat_h */
```

<fcntl.h>

The following defines are in <fcntl.h>:

```
#ifndef __fcntl_h
#define __fcntl_h

/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#if defined(_SYSV_SOURCE) || defined(_POSIX_SOURCE)
#ifndef __sys_file_h
#define     O_RDONLY 0
#define     O_WRONLY 1
#define     O_RDWR   2
#define     O_APPEND 0x08     /*append (writes guaranteed at
                               the end)*/

/* Flag values accessible only to open(2) */
#define  O_CREAT  0x100  /* open with file creat
                          (uses third open arg)*/
#define  O_TRUNC  0x200  /* open with truncation */
#define  O_EXCL   0x400  /* exclusive open */

/* fcntl(2) requests */
#define  F_DUPFD    0  /* Duplicate fildes */
#define  F_GETFD    1  /* Get fildes flags */
#define  F_SETFD    2  /* Set fildes flags */
#define  F_GETFL    3  /* Get file flags */
#define  F_SETFL    4  /* Set file flags */
#define  F_GETLK    5  /* Get file lock */
#define  F_SETLK    6  /* Set file lock */
#define  F_SETLKW   7  /* Set file lock and wait */
#endif /* !__sys_file_h */
```

```
/* file segment locking set data type */
/* - information passed to system by user */
struct flock {
            short         l_type;
            short         l_whence;
            long          l_start;
            long          l_len; /* len = 0 means
                          until end of file */
#ifdef _POSIX_SOURCE
            pid_t         l_pid;
#else /* !_POSIX_SOURCE */
            int           l_pid; #
endif /* _POSIX_SOURCE */
};

/* file segment locking types */
#define   F_RDLCK  01   /* Read lock */
#define   F_WRLCK  02   /* Write lock */
#define   F_UNLCK  03   /* Remove lock(s) */

#endif /* _SYSV_SOURCE || _POSIX_SOURCE */

#ifdef _SYSV_SOURCE
/* Historical flag values accessible to open(2) and
fcntl(2) */
#define     O_NDELAY 04 /* Non-blocking I/O */
#endif /* _SYSV_SOURCE */

#ifdef _BSD_SOURCE
/* Additional fcntl(2) requests */
#define   F_GETOWN   8   /* Get owner */
#define  F_SETOWN   9   /* Set owner */
#endif /* _BSD_SOURCE */
```

(continued)➡

```
#ifdef _POSIX_SOURCE
/* File access mode mask */
#define     O_ACCMODE 03

/* POSIX-defined argument to F_SETFD */
#define     FD_CLOEXEC    0x0001
/*
 * POSIX-defined flag values accessible
 * to open(2) and/or fcntl(2)
*/
#define  O_NONBLOCK  0x00004000 /* O_NDELAY POSIX style */
#define  O_NOCTTY    0x00008000 /* don't assign
                                 controlling tty */
#endif /* _POSIX_SOURCE */

#ifdef _AUX_SOURCE
/* Implementation-defined flag values accessible to open(2) */
#define O_GETCTTY   0x00010000 /* force controlling
                                 tty assignment */
#define O_GLOBAL    0x80000000 /* force allocation from
                                 global table */
#endif /* _AUX_SOURCE */

#endif /* !__fcntl_h */
```

<limits.h>

The header file `<limits.h>` contains the following constants:

```
#ifndef __limits_h
#define __limits_h

/*
 * These symbolic names are defined in the SVID, ANSI C,
 * POSIX, and/or intro(2).
 */
```

```
/*
 * sizes of integral types; constants defined by ANSI C
 */
/* number of bits in a char */
#define     CHAR_BIT      8

/* max integer value of a char */
#define     SCHAR_MAX     127
#define     UCHAR_MAX     255
#define     CHAR_MAX      SCHAR_MAX

/* min integer value of a char */
#define     SCHAR_MIN     -128
#define     CHAR_MIN SCHAR_MIN

/* max decimal value of a long */
#define     LONG_MAX      2147483647

/* min decimal value of a long */
#define     LONG_MIN      -2147483648

/* max decimal value of a short */
#define     SHRT_MAX      32767

/* min decimal value of a short */
#define     SHRT_MIN      -32768

/* max decimal value of an int */
#define     INT_MAX       LONG_MAX

/* min decimal value of an int */
#define     INT_MIN       LONG_MIN

/* max decimal value of an unsigned long */
#define     ULONG_MAX     4294967295

/* max decimal value of an unsigned short */
#define     USHRT_MAX     65535
```

(continued)➡

```
/* max decimal value of an unsigned int */
#define     UINT_MAX      ULONG_MAX

/* max number of bytes in multibyte character, */
/* for any supported locale */
#define     MB_LEN_MAX     2

/*
 * operating system constants and other numeric constants
 * not defined by ANSI C; where applicable, cross-referenced
 * to constant and/or file used internally; configurable by
 * kconfig(1m) values are also cross-referenced to <sys/var.h>
 * and uvar(2)
 */

#ifdef _SYSV_SOURCE

/* max number of processes per user id;
cf. MAXUP <sys/config.h>, v.v_maxup <sys/var.h> */
#define     CHILD_MAX      25

/* max size of a file in bytes; see ULIMIT below */
#define   FCHR_MAX   0   /* overflow! */

/* number of bits in a long */
#define     LONG_BIT      WORD_BIT

/* max decimal value of a double */
#define     MAXDOUBLE     1.79769313486231470e+308

/* max number of bytes in terminal input line;
cf. CANBSIZ <sys/param.h> */
#define     MAX_CHAR      256

/* max number of characters in a file name;
cf. SVFSDIRSIZ <svfs/fsdir.h> */
#define     NAME_MAX                        14

/* max value for a process ID; cf. MAXPID <sys/param.h> */
#define     PID_MAX                         30000
```

```
/* max number of bytes written to a pipe in a write;
PIPSIZ <svfs/inode.h> */
#define     PIPE_MAX      5120

/* max number of processes system-wide;
cf. NPROC <sys/config.h>, v.v_proc <sys/var.h> */
#define     PROC_MAX      50

/* number of bytes in a physical I/O block;
  cf. DEV_BSIZE <sys/param.h> */
#define     STD_BLK                                 512

/* number of chars in uname(2) strings; cf. <sys/utsname.h> */
#define     SYS_NMLN       9

/* max number of open files system-wide;
cf. NFILE <sys/config.h>, v.v_file <sys/var.h> */
#define     SYS_OPEN       100

/* max number of unique names generated by tmpnam(3) */
#define     TMP_MAX                               17576

/* max value for a user or group ID;
  cf. MAXUID <sys/param.h> */
#define    UID_MAX    60000

/* max decimal value of an unsigned int */
#define    USI_MAX    ULONG_MAX

/* number of bits in a word (int) */
#define    WORD_BIT       32
#endif /* _SYSV_SOURCE */

#ifdef _POSIX_SOURCE
/*
 * minimum values for implementation-specific
 * constants defined by POSIX.1
 */
```

(continued)➠

```
#define     _POSIX_ARG_MAX         4096
#define     _POSIX_CHILD_MAX       6
#define     _POSIX_LINK_MAX        8
#define     _POSIX_MAX_CANON       255
#define     _POSIX_MAX_INPUT       255
#define     _POSIX_NAME_MAX        14
#define     _POSIX_NGROUPS_MAX     0
#define     _POSIX_OPEN_MAX        16
#define     _POSIX_PATH_MAX        255
#define     _POSIX_PIPE_BUF        512

#ifdef      CHILD_MAX
#undef      CHILD_MAX
#endif      /* CHILD_MAX */
/*
```

POSIX requires indeterminate values to be omitted. `NAME_MAX` is file-system dependent; use `pathconf()` to obtain a value for `NAME_MAX`.

```
 */
#ifdef       NAME_MAX
#undef       NAME_MAX
#endif       /* NAME_MAX */
/* max number of supplementary group IDs; */
/* cf. NGROUPS <sys/param.h> */
#define       NGROUPS_MAX  8
/* max number of bytes in canonical input line; */
/* cf. CANBSIZ <sys/param.h> */
#define       MAX_CANON    256
/* max number of bytes in terminal input queue; */
/* cf. CANBSIZ <sys/param.h> */
#define       MAX_INPUT    256
#endif /* _POSIX_SOURCE */

#if defined(_SYSV_SOURCE) || defined(_POSIX_SOURCE)
/*
```

```
/* max length of arguments to exec(2); */
/* cf. NCARGS <sys/param.h> */
#define     ARG_MAX        5120

/* max number of links per file; */
/* cf. MAXLINK <sys/param.h> */
#define     LINK_MAX       1000

/* max number of open files per process; */
/* cf. NOFILE <sys/param.h> */
#define     OPEN_MAX       32

/* max number of characters in a path name; */
/* cf. MAXPATHLEN <sys/param.h> */
#define     PATH_MAX       1024
/* max number of bytes atomic in write to a pipe; */
/* PIPSIZ <svfs/inode.h> */
#define     PIPE_BUF                             5120
#endif /* _SYSV_SOURCE || _POSIX_SOURCE */

#if defined(_SYSV_SOURCE) || defined(_FIPS_151_SOURCE)
/*
 * CLK_TCK is also defined in <time.h>,
 * for historical reasons.
 * number of clock ticks per second;
 * cf. HZ and CLKTICK <sys/param.h>
 */
#ifndef CLK_TCK
#define     CLK_TCK                              60
#endif /* !CLK_TCK */
/* max number of characters in a password */
#define     PASS_MAX       8
/* max value for a process ID; cf. MAXPID <sys/param.h> */
#define     PID_MAX                              30000
```

(continued)➡

```
* max value for a user or group ID; cf. MAXUID <sys/param.h>
*/
#define     UID_MAX                                60000
#endif /* _SYSV_SOURCE || _FIPS_151_SOURCE */

#ifdef _AUX_SOURCE

/* max size of file in 512-byte blocks; */
/* cf. CDLIMIT <sys/param.h>, ulimit(2) */
#define     ULIMIT                                 16777216
#endif /* _AUX_SOURCE */

#endif /* !__limits_h */
```

<utime.h>

The header file `<utime.h>` defines the `utimbuf` structure for use with `utime`(2P):

```
struct utimbuf {
            time_t actime;
            time_t modtime;
};
```

Migrating programs from A/UX to A/UX POSIX

In this section, several examples are presented illustrating differences between the standard A/UX environment and the A/UX POSIX environment. In the first example, identical signal-catching behavior is obtained in both environments. In the second, identical terminal interface setup is shown. In the third and fourth, determining the values of system variables is demonstrated.

Manipulate signal sets

The following code fragment shows how a program uses 4.2BSD signals to set up an interrupt-handling routine for `SIGINT` and block `SIGQUIT` signals, while handling a `SIGINT` signal.

```
#include <signal.h>

int interrupt();

struct sigvec s;
struct sigvec os;

s.sv_handler = interrupt;
s.sv_mask = sigmask(SIGQUIT);
s.sv_onstack = 0;

if (sigvec(SIGINT, &s, &os) == -1)
            perror("sigvec");
interrupt()
{
            printf("Interrupt\n");
}
```

The same example, using A/UX POSIX signal functions:

```
#include <signal.h>

extern void interrupt();

struct sigaction action;
struct sigaction oldaction;

if (sigemptyset(&action.sa_mask) == -1)
            perror("siginitset");

if (sigaddset(&action.sa_mask, SIGQUIT) == -1)
            perror("sigaddset");
action.sa_handler = interrupt;
action.sa_flags = 0;
if (sigaction(SIGINT, &action, &oldaction) == -1)
            perror("sigaction");
}

void
interrupt()
{
            printf("Interrupt\n");
}
```

Terminal control

`tcgetattr`(3P) and `tcsetattr`(3P) are used to get and set terminal attributes. Previously, `ioctl`(2) was used for getting and setting terminal attributes.

`tcgetattr`

Using `ioctl`(), a program gets the value of the suspend character as follows:

```
#include <sys/ioctl.h>

struct ltchars lc;
char suspend;
```

```
if (ioctl(0, TIOCGLTC, &lc) == -1)
            return(-1);
suspend = lc.t_suspc;
```

In the A/UX POSIX environment, `tcgetattr`() is used:

```
#include <unistd.h>
#include <termios.h>

struct termios t;
char suspend;

if (tcgetattr(STDIN_FILENO, &t) == -1)
            perror("tcgetattr");
suspend = t.c_cc[VSUSP];
```

`tcsetattr`

The following code fragment sets the `TOSTOP` flag for a process using `ioctl`(2):

```
#include <sys/termio.h>
#include <sys/ioctl.h>

int compat;

compat = TOSTOP;
if (ioctl(1, TIOCSCOMPAT, &compat) == -1)
            perror("compat");
```

In the POSIX environment, `tcsetattr`() is used to set the `TOSTOP` flag as follows:

```
#include <unistd.h>
#include <termios.h>

struct termios t;

if (tcgetattr(STDOUT_FILENO, &t) == -1)
            perror("tcgetattr");

t.c_lflag |= TOSTOP;

if (tcsetattr(STDOUT_FILENO, TCSANOW, &t) == -1)
            perror("tcsetattr");
```

Configurable system variables

New functionality is provided in the POSIX environment to query system and file-related variables. The three routines, `sysconf()`, `pathconf()`, and `fpathconf()` provide this functionality.

`fpathconf`

The following example uses `fpathconf()` to determine the size of a buffer used to hold pathnames:

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <limits.h>

long i;
int fd;
har *buf;
char *malloc();

if ((fd = open("./file", O_RDWR)) == -1)
            perror("open");
if ((i = fpathconf(fd, _PC_PATH_MAX)) == -1)
            perror("fpathconf");
if ((buf = malloc((unsigned) i)) == NULL)
            perror("malloc");
```

sysconf

The following example uses `sysconf()` to allocate space for a table to keep track of child processes:

```
#include <stdio.h>
#include <unistd.h>
#include <limits.h>

struct cldp {
        int pid;
        int info;
};

struct cldp *buf;
long i;
char *malloc();

i = sysconf(_SC_CHILD_MAX);
if (((char *)buf =
  malloc((unsigned) (i * sizeof(struct cldp)))) == NULL)
            perror("malloc");
```

Appendix C: A/UX Guide to POSIX

1. General

This appendix describes the Apple A/UX implementation of the IEEE Standard 1003.1-1990, Portable Operating System Interface for Computer Environments, also known as POSIX.1. POSIX.1 was approved by the IEEE Standards Board on September 28, 1990. POSIX.1 is also an international standard, ISO 9945-1, dated December 7, 1990. The A/UX implementation also complies with the United States Federal Information Processing Standard (FIPS) 151-1, which is equivalent to POSIX.1 with certain implementation options required.

1.3 Conformance

1.3.1 Implementation conformance

1.3.1.1 Requirements

A/UX provides a complete POSIX environment. When compiling conforming POSIX applications, however, special flags or options must be used with the compiler in order to provide the application with the POSIX environment; once the application is compiled, the application is executed normally.

Conforming POSIX applications must be compiled by invoking the A/UX C compiler with a special option, `-ZP`, and specifying the A/UX POSIX library, `/lib/libposix.a`, as in this example:

```
cc -ZP sample.c -lposix
```

These options cause the compiler front end to arrange for the following. The `-ZP` option will cause the C preprocessor, `cpp`(1), to be invoked with the feature test macro `_POSIX_SOURCE` defined and other A/UX feature test macros disabled; section 2.7.2, "POSIX.1 Symbols." To allow a POSIX application to use these A/UX extensions, replace this option with an explicit definition of `_POSIX_SOURCE`, as shown here:

```
cc -D_POSIX_SOURCE sample.c -lposix
```

The `-lposix` option instructs the link editor, `ld`(1), to link the A/UX POSIX library before the standard C library, `/libc/libc.a`. This is necessary since the A/UX POSIX library contains some functions that also exist in the standard C library. The POSIX library also arranges for the application startup code to set the A/UX POSIX process compatibility flags; see `setcompat`(2).

The POSIX library should be specified on the command line before any additional libraries required by the application. For example, you might use the following command to link your application with the shared version of the standard C library in order to reduce the size of the compiled application on disk:

```
cc -D_POSIX_SOURCE sample.c -lposix -lc_s
```

For additional information about these options, see `cc`(1).

1.3.1.2 Documentation

This document fulfills the documentation requirements in section 1.3.1.2 of POSIX.1 and describes the behavior that is implementation defined or where the standard specifies that implementations may vary. This document, along with POSIX.1, fully describes the A/UX POSIX implementation. The format and organization of this document follow POSIX.1.

◆ **Note** Strictly conforming POSIX.1 applications may not depend on **unspecified** or **implementation-defined** behavior described in this document. Applications that depend on such behavior may be unportable to other POSIX environments and/or future versions of this implementation. ◆

1.3.1.3 Conforming implementation options

A/UX supports all three implementation options described in section 1.3.1.3 of POSIX.1.

`{NGROUPS_MAX}`	multiple groups option
`{_POSIX_JOB_CONTROL}`	job control option
`{_POSIX_CHOWN_RESTRICTED}`	administrative/security option

1.3.3 Language-dependent services for the C programming language

A/UX conforms to POSIX.1, C Language Binding (Common-Usage C Language-Dependent System Support).

2. Terminology and general requirements

2.1 Conventions

This document uses the following typographic conventions.

1. The *italic* font is used for:
   - □ Symbolic parameters that are substituted with actual values by the application, for example, *path*.
2. The **bold** font is used to represent environment variables, for example, **PATH,** and cross-references to defined terms, for example, **file.**
3. The constant-width (`Courier`) font is used for:
   - □ Examples of system input or output where exact usage is depicted.
   - □ C language data types and function names, for example, `open()`.
   - □ Global external variable names, for example, `errno`.
   - □ References to symbolic constants and C language header files.
   - □ References to the A/UX system documentation, the A/UX Programmers and Administrators References, for example, `init`(1m); the relevant section number is indicated within the parentheses.

2.2 Definitions

2.2.2 General terms

2.2.2.4 Appropriate privileges

If a function call requires **appropriate privileges,** the effective user ID of the calling process must be zero.

2.2.2.27 File

In addition to the **file** types specified by POSIX.1, A/UX supports two additional file types: symbolic links and sockets; these types are defined in `<sys/stat.h>` as `S_IFLNK` and `S_IFSOCK`, respectively. See `symlink`(2) and `socket`(2) for more information.

2.2.2.30 File group class

No other members of a **file group class** are defined.

2.2.2.55 Parent process ID

After the creator's lifetime has ended, the **parent process ID** of the created process is the process ID of `init`(1M).

2.2.2.57 Pathname

A pathname that begins with two successive slashes is interpreted as if it begins with a single slash.

2.2.2.69 Read-only file system

A read-only file system is specified using `mount`(2) or `mount`(1M).

2.2.2.83 Supplementary group ID

A process's effective group ID is included in its list of supplementary group IDs.

2.3 General concepts

2.3.1 Extended security controls

There are no extended security controls.

2.3.2 File access permissions

There are no additional file access permissions or alternative access methods.

2.3.5 File times update

The A/UX extensions `fsync`(2), `ftruncate`(2), `mknod`(2), `symlink`(2) and `truncate`(2) will change the values of `st_atime`, `st_mtime`, and `st_ctime`.

2.4 Error numbers

The following additional error numbers are defined in `<errno.h>`.

`ENOTBLK`	block device required
`ETXTBSY`	text file busy
`ENOMSG`	no message of desired type
`EIDRM`	identifier removed
`ECHRNG`	channel number out of range
`EL2NSYNC`	level 2 not synchronized

`EL3HLT`	level 3 halted
`EL3RST`	level 3 reset
`ELNRNG`	link number out of range
`EUNATCH`	protocol driver not attached
`ENOCSI`	no CSI structure available
`EL2HLT`	level 2 halted
`EWOULDBLOCK`	operation would block
`EINPROGRESS`	operation now in progress
`EALREADY`	operation already in progress
`ENOTSOCK`	socket operation on a nonsocket
`EDESTADDRREQ`	destination address required
`EMSGSIZE`	message too long
`EPROTOTYPE`	protocol wrong type for socket
`ENOPROTOOPT`	protocol not available
`EPROTONOSUPPORT`	protocol not supported
`ESOCKTNOSUPPORT`	socket type not supported
`EOPNOTSUPP`	operation not supported on socket
`EPFNOSUPPORT`	protocol family not supported
`EAFNOSUPPORT`	address family not supported by protocol
`EADDRINUSE`	address already in use
`EADDRNOTAVAIL`	can't assign requested address
`ENETDOWN`	network is down
`ENETUNREACH`	network is unreachable
`ENETRESET`	network dropped connection on reset
`ECONNABORTED`	software caused connection abort
`ECONNRESET`	connection reset by peer
`ENOBUFS`	no buffer space available
`EISCONN`	socket is already connected
`ENOTCONN`	socket is not connected
`ESHUTDOWN`	can't send after socket shutdown

`ETOOMANYREFS`	too many references
`ETIMEDOUT`	connection timed out
`ECONNREFUSED`	connection refused
`ELOOP`	too many levels of symbolic links
`EHOSTDOWN`	host is down
`EHOSTUNREACH`	no route to host
`ENOSTR`	device not a stream
`ENODATA`	no data (for no delay I/O)
`ETIME`	timer expired
`ENOSR`	out of streams resources
`ESTALE`	stale NFS file handle
`EREMOTE`	too many levels of remote in path
`EPROCLIM`	too many processes
`EUSERS`	too many users
`EDQUOT`	disk quota exceeded
`EDEADLOCK`	locking deadlock error
`ENOLCK`	no record locks available

`EFAULT` will be generated when an invalid address is passed as an argument to a function.

`EFBIG` will never occur if the system default value of the per-process file size limit is used; see `ulimit`(2) and `ULIMIT` in `<limits.h>`. This file size limit is expressed in 512-byte blocks; in terms of bytes, the default limit is greater than `ULONG_MAX`, thus `ULONG_MAX` will be exceeded before the file size limit is reached. A process may change its file size limit using `ulimit`(2).

2.5 Primitive system data types

The following additional implementation-defined data types are defined in `<sys/types.h>`:

```
typedef     char *                   caddr_t;
typedef     short                    cnt_t;
typedef     long                     daddr_t;
typedef     long                     key_t;
typedef     int                      label_t[13];
typedef     long                     paddr_t;
typedef     long                     ubadr_t;
typedef     unsigned char            uchar_t;
typedef     unsigned int             uint_t;
typedef     unsigned long            ulong_t;
typedef     unsigned short           ushort_t;
```

2.6 Environment description

A/UX allows the use of the entire character set for environment variable names. However, as recommended in this section of POSIX.1, applications should only create names that are composed solely of characters from the **portable filename character set.**

2.7 C Language definitions

2.7.2 POSIX.1 symbols

In addition to `_POSIX_SOURCE`, the A/UX C language header files use the feature test macros `_SYSV_SOURCE`, `_BSD_SOURCE`, and `_AUX_SOURCE`. These macros control the visibility of symbols specified by or derived from the historical implementations on which the A/UX POSIX environment is based.

The feature test macro `_FIPS_151_SOURCE` is also used to provide backwards compatibility with the original version of the POSIX FIPS, which was based on Draft 12 of POSIX.1, dated October 12, 1987. `_FIPS_151_SOURCE` is obsolete and its use in new applications is strongly discouraged.

2.8 Numerical limits

These magnitude limitations are fixed by A/UX in `<limits.h>`:

Name	*Value*
`{NGROUPS_MAX}`	8
`{ARG_MAX}`	5120
`{CHILD_MAX}`	*
`{OPEN_MAX}`	32
`{STREAM_MAX}`	32

{TZNAME_MAX}	50
{LINK_MAX}	32767
{MAX_CANON}	256
{MAX_INPUT}	256
{NAME_MAX}	*
{PATH_MAX}	1024
{PIPE_BUF}	5120
{SSIZE_MAX}	2147483647

*CHILD_MAX and NAME_MAX are run-time variant. Applications may query these values using sysconf.

2.9 Symbolic constants

The following symbolic constants are defined in <unistd.h>.

Values for the mode argument to access(2):

R_OK	4	read permission
W_OK	2	write permission
X_OK	1	execute or search permission
F_OK	0	existence only

Values for the whence argument to lseek(2):

SEEK_SET	0	beginning of file
SEEK_CUR	1	current position
SEEK_END	2	end of file

POSIX flag options:

{_POSIX_JOB_CONTROL}	1
{_POSIX_CHOWN_RESTRICTED}	1
{_POSIX_SAVED_IDS}	1
{_POSIX_NO_TRUNC}	1
{_POSIX_V_DISABLE}	0377
{_POSIX_VERSION}	199009L

3. Process primitives

3.1 Process creation and execution

3.1.1 Process creation

Function: `fork()`

3.1.1.2 Description

These additional process characteristics are inherited by the child process:

- process compatibility flags
- profiling status
- scheduling parameters, see `nice`(2)
- all attached shared memory segments
- trace flag
- mapped regions of physical memory, see `phys`(2)
- resource limits, see `ulimit`(2)

Open directory stream positioning is shared with the parent process. Process locks, text locks, and data locks are not inherited by the child. The *semadj* values are cleared for the child process.

3.1.2 Execute a file

Functions: `execl()`, `execv()`, `execle()`, `execve()`, `execlp()`, `execvp()`

3.1.2.2 Description

If the `PATH` environment variable is undefined, the directories searched to find the file are `/bin` and `/usr/bin`.

The number of bytes available for a new process's argument and environment lists, `ARG_MAX`, includes null terminators.

If the `exec` function failed, but was able to locate the process image, the `st_atime` field is marked for update.

3.1.2.4 Errors

A/UX will detect an error and set `errno` to `EACCES` if a process attempts to execute a file that is not a regular file.

A/UX will detect an error and set `errno` to `ENOMEM` if the new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

3.2 Process termination

3.2.1 Wait for process termination

Functions: `wait()`, `waitpid()`

3.2.1.2 Description

If a parent process terminates without waiting for its children to terminate, the children will be assigned the parent process ID 1. This process ID corresponds to the initialization process, `init`(1M).

If the child process is stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop, and the low order 8 bits will be set equal to 0177.

3.2.2 Terminate a process

Function: `_exit()`

3.2.2.2 Description

If a parent process terminates without waiting for its children to terminate, the children will be assigned the parent process ID 1. This process ID corresponds to the initialization process, `init`(1M).

3.3 Signals

3.3.1 Signal concepts

3.3.1.1 Signal names

The following additional signals are defined in `<signal.h>`.

`SIGTRAP`	trace trap
`SIGIOT`	IOT instruction
`SIGEMT`	EMT instruction
`SIGBUS`	bus error
`SIGSYS`	bad argument to a system call
`SIGPWR`	power-fail restart
`SIGVTALRM`	virtual time alarm
`SIGPROF`	profiling timer alarm
`SIGWINCH`	window size change
`SIGURG`	urgent condition present on socket
`SIGIO`	I/O is possible on a descriptor

3.3.1.2 Signal generation and delivery

If there is a subsequent instance of a pending signal, the signal will be delivered only once.

Under the following conditions, which are not specified by POSIX.1, A/UX may generate the following signals.

`SIGTRAP`	trace/breakpoint trap
`SIGBUS`	bus error
`SIGSYS`	bad argument to a system call
`SIGPWR`	power-fail restart
`SIGVTALRM`	virtual time alarm
`SIGPROF`	profiling timer alarm
`SIGWINCH`	window size change
`SIGURG`	urgent socket condition

3.3.1.3 Signal actions

If a process ignores `SIGSEGV` or `SIGILL`, the instruction is restarted. If a process ignores `SIGFPE`, the offending instruction is skipped.

If the action for `SIGCHLD` is set to `SIG_IGN`, the behavior is as if the action were set to `SIG_DFL`.

There are three arguments to signal catching functions: a signal number, a code, and a pointer to a `sigcontext` structure; this structure is defined in `<sys/signal.h>`.

If a signal-catching function for `SIGSEGV` or `SIGILL` returns normally, the instruction is restarted. If a signal-catching function for `SIGFPE` returns normally, the offending instruction is skipped.

Establishing a signal-catching function for `SIGCHLD` while a process has child processes is permitted.

3.3.2 Send a signal to a process

Function: `kill()`

3.3.2.2 Description

The `kill()` function excludes certain processes from a selected list of processes; these "system processes" are process ID 0 and 1.

If *pid* is -1, the behavior of the `kill()` function is as follows. If the sending process has appropriate privileges, the signal will be sent to all processes, excluding the system processes.

3.3.3 Manipulate signal sets

Functions: `sigaddset()`, `sigdelset()`, `sigismember()`

3.3.3.4 Errors

The functions `sigaddset()`, `sigdelset()`, and `sigismember()` detect the condition that sets `errno` to `EINVAL`.

3.3.4 Examine and change signal action

Function: `sigaction()`

3.3.4.2 Description

These additional flag bits for `sa_flags` of the `sigaction` structure are defined in `<signal.h>`:

`SA_ONSTACK`	take signal on signal stack
`SA_INTERRUPT`	do not restart system call on signal return

If the previous action for sig had been established by the `signal()` function, the values of the fields in the structure pointed to by oact will be as follows: the `sa_mask` and `sa_flags` fields will be set to zero and the `sa_handler` field will contain the address of the handler that was originally passed to the `signal()` function.

3.3.4.4 Errors

An attempt to set the `SIG_DFL` action for a signal that cannot be caught or ignored will cause the return value to be set to -1 and `errno` to be set to `EINVAL`.

3.3.6 Examine pending signals

Function: `sigpending()`

3.3.6.4 Errors

If *set* points to an invalid address, `sigpending()` will return -1 and set `errno` to `EFAULT`.

4. Process environment

4.2 User identification

4.2.4 Get user name

Function: `getlogin()`

4.2.4.3 Returns

The return value points to static data whose content is overwritten by each call.

4.2.4.4 Errors

There are no error conditions for `getlogin()` other than the user name not being found.

4.4 System identification

4.4.1 Get system name

Function: `uname()`

4.4.1.2 Description

The `utsname` structure is defined in `<sys/utsname.h>`:

```
struct        utsname {

              char          sysname[9];
              char          nodename[9];
              char          release[9];
              char          version[9];
              char          machine[9];
};
```

The values for members of `utsname` are string constants defined at the time the system is created or initiated.

4.4.1.4 Errors

If *name* points to an invalid address, `uname()` will return -1 and set `errno` to `EFAULT`.

4.5 Time

4.5.1 Get system time

Function: `time()`

4.5.1.4 Errors

If *tloc* points to an illegal address, `time()` will return -1 and set `errno` to `EFAULT`.

4.5.2 Get process time

Function: `times()`

4.5.2.2 Description

There are no additional members of the *tms* structure in `<sys/times.h>`.

4.5.2.4 Errors

If *buffer* points to an illegal address, `times()` will return -1 and set `errno` to `EFAULT`.

4.6 Environment variables

4.6.1 Environment access

Function: `getenv()`

4.6.1.3 Returns

The return value from `getenv()` points to the environment list, not static data.

4.6.1.4 Errors

There are no error conditions for `getenv()` other than the environment variable not being found.

4.7 Terminal identification

4.7.1 Generate terminal pathname

Function: `ctermid()`

4.7.1.3 Returns

If *s* is a NULL pointer, the string is generated in static data that may be overwritten by a subsequent call to `ctermid`().

4.7.1.4 Errors

There are no error conditions for `ctermid()`.

4.7.2 Determine terminal device name

Function: `ttyname()`, `isatty()`

4.7.2.2 Description

The return value of `ttyname()` points to static data that is overwritten by each call.

4.7.2.4 Errors

There are no error conditions for `ttyname()` or `isatty()` other than *fildes* not describing a terminal device.

4.8 Configurable system variables

4.8.1 Get configurable system variables

4.8.1.2 Description

A/UX supports the following additional configurable system variables. These variables are supported for backwards compatibility only; their use in new applications is strongly discouraged. The macros listed below are only defined if the feature test macro `_FIPS_151_SOURCE` is defined. For definitions of these variables, see Draft 12 of POSIX.1, dated October 12, 1987.

Variable	*Name value*
`PASS_MAX`	`_SC_PASS_MAX`
`PID_MAX`	`_SC_PID_MAX`
`UID_MAX`	`_SC_UID_MAX`
`_POSIX_EXIT_SIGHUP`	`_SC_EXIT_SIGHUP`

5. Files and directories

5.1 Directories

5.1.1 Format of directory entries

A/UX supports two file-system types on local storage media: System V and Berkeley. Each directory is a file that contains one entry for each file contained in the directory. In System V file systems, directory entries are defined by the structure `svfsdirect` in `<svfs/fsdir.h>`:

```
#define      SVFSDIRSIZ      14

struct       svfsdirect {
             ino_t        d_ino;
             char         _name[SVFSDIRSIZ];
};
```

For Berkeley file systems, directory entries are defined by the structure direct in `<ufs/fsdir.h>`:

```
#define      MAXNAMLEN    255

struct       direct {
             u_long       d_fileno;
             u_short      d_reclen;
             u_short      d_namlen;
             char         d_name[MAXNAMLEN + 1];
};
```

The `dirent` structure in `<dirent.h>` has three elements in addition to `d_name`:

```
#define      _SYS_NAME_MAX            255

struct       dirent {
             u_long       d_fileno;
             u_short      d_reclen;
             u_short      d_namlen;
             char         d_name[_SYS_NAME_MAX + 1];

};
```

5.1.2 Directory operations

5.1.2.2 Description

The type `DIR` is implemented on top of a file descriptor.

The pointer returned by `readdir()` points to data that may be overwritten by a subsequent call on the same directory stream.

The `readdir()` function buffers several directory entries per actual read operation.

Upon return from the `closedir()` function, the value of *dirp* no longer points to an accessible object of type `DIR`.

5.1.2.4 Errors

A/UX detects the conditions that would cause `opendir` to set `errno` to the values `EMFILE` and `ENFILE`.

This implementation detects the condition that would cause `readdir` to set `errno` to the value `EBADF`.

A/UX detects the condition that would cause `closedir` to set `errno` to the value `EBADF`.

5.3 General file creation

5.3.1 Open a file

Function: `open()`

5.3.1.2 Description

The following additional flags for `oflag` are defined in `<fcntl.h>`:

`O_NDELAY`	request System V, nonblocking I/O
`O_GETCTTY`	force assignment of the controlling terminal
`O_GLOBAL`	allocate descriptor from global file table

If bits in mode other than file permissions are used, the permissions on the file will be undefined.

If `open()` is called with `O_EXCL`, `O_CREAT` must also be present; otherwise `O_EXCL` will be ignored.

5.3.3 Set file creation mask

Function: `umask()`

5.3.3.2 Description

A/UX defines one additional bit in the type `mode_t`, which may be masked with the file mode creation mask. `S_ISVTX`, defined in `<sys/stat.h>`, indicates to the system that memory regions associated with this file should not be released after use.

5.3.4 Link to a file

Function: `link()`

5.3.4.2 Description

A/UX does not support linking across file systems. The A/UX extension `symlink(2)` provides this functionality.

Processes with appropriate privileges may use `link()` on directories.

The calling process is not required to have permission to access the existing file.

5.4 Special file creation

5.4.1 Make a directory

Function: `mkdir()`

5.4.1.2 Description

If bits in *mode* other than file permissions are used, the permissions on the directory will be undefined.

In the A/UX POSIX environment, the directory's group ID shall be set to the group ID of the directory in which the directory is being created.

5.4.2 Make a FIFO special file

Function: `mkfifo()`

5.4.2.2 Description

If bits in *mode* other than file permissions are used, the permissions on the `FIFO` special file will be undefined.

5.5 File removal

5.5.1 Remove directory entries

5.5.1.2 Description

Processes with appropriate privileges may use `unlink()` on directories.

5.5.2 Remove a directory

Function: `rmdir()`

5.5.2.2 Description

If an attempt is made to remove the root directory, `rmdir()` will return -1 and set `errno` to `EBUSY`.

If an attempt is made to remove the current working directory, `rmdir()` will return -1 and set `errno` to `EINVAL`.

5.6 File characteristics

5.6.1 File characteristics: Header and data structure

The stat struct ire is defined in `<sys/stat.h>`:

```
struct stat {

            dev_t       st_dev;
            ino_t       st_ino;
            mode_t      st_mode;
            nlink_t     st_nlink;
            uid_t       st_uid;
            gid_t       st_gid;
            dev_t       st_rdev;
            off_t       st_size;
            time_t      st_atime;
            time_t      st_mtime;
            time_t      st_ctime;
            long        st_blksize;
            long        st_blocks;
};
```

`st_rdev` is defined only for block or character devices. For these devices, `st_rdev` specifies the device ID.

5.6.1.2 `<sys/stat.h>` file modes

No other bits are included in `S_IRWXU`, `S_IRWXG`, and `S_IRWXO`.

5.6.2 Get file status

Functions: `stat(), fstat()`

5.6.2.2 Description

This implementation does not provide any additional or alternate file access controls.

5.6.2.4 Errors

This implementation detects the following additional error conditions for `stat();`

`EFAULT`	*buf* or *path* points to an invalid address
`ELOOP`	too many symbolic links were encountered in translating a pathname

5.6.3 Check file accessibility

Function: `chmod()`

5.6.3.2 Description

A process with appropriate privileges is always granted execute permission even though execute permission is meaningful only for directories and regular files, and `exec` requires that at least one execute mode bit be set for regular files to be executable.

5.6.3.4 Errors

The A/UX implementation of `access()` detects the condition that sets `errno` to `EINVAL`.

5.6.4 Change file modes

Function: `chmod()`

5.6.4.2 Description

`S_ISUID` and `S_ISGID` bits may be ignored if the owner is the superuser and the file system is a remotely mounted file system.

`chmod()` of an open file has no effect on the open file descriptor(s).

5.6.5 Change owner and group of a file

Function: `chown()`

5.6.5.2 Description

If a process with appropriate privileges performs a `chown()`, the `S_ISUID` and `S_ISGID` bits are not changed.

5.7 Configurable pathname variables

5.7.1 Get configurable pathname variables

5.7.1.2 Description

A/UX supports the following additional configurable pathname variables. These variables are supported for backwards compatibility only; their use in new applications is strongly discouraged. The macros listed below are only defined if the feature test macro

`_FIPS_151_SOURCE` is defined. For definitions of these variables, see Draft 12 of POSIX.1, dated October 12, 1987.

Variable	*Name value*
`_POSIX_CHOWN_SUP_GRP`	`_PC_CHOWN_SUP_GRP`
`_POSIX_DIR_DOTS`	`_PC_DIR_DOTS`
`_POSIX_GROUP_PARENT`	`_PC_GROUP_PARENT`
`_POSIX_UTIME_OWNER`	`_PC_UTIME_OWNER`

A/UX does not impose any restrictions on the association of variable names with file types for which the limit or option may not be relevant.

5.7.1.4 Errors

This implementation detects the conditions that set `errno` to the following values: `EACCES`, `ENAMETOOLONG`, `ENOENT`, `ENOTDIR`, and `EBADF`.

6. Input and output primitives

6.4 Input and output

6.4.1 Read from a file

Function: `read()`

6.4.1.2 Description

If `read()` is interrupted by a signal after successfully reading some data, it will return the number of bytes read.

After end-of-file is reached, subsequent `read()` requests on a device special file will return zero.

If *nbyte* is greater than `INT_MAX`, `read()` will return -1 and set `errno` to `EINVAL`.

6.4.1.4 Errors

The error `EIO` will be reported if a physical hardware error occurs.

6.4.2 Write to a file

Function: `write()`

6.4.2.2 Description

If `write()` is interrupted by a signal after successfully writing some data, it will return the number of bytes written.

If *nbyte* is greater than `INT_MAX`, `write()` will return -1 and set `errno` to `EINVAL`.

6.4.2.4 Errors

`EFBIG` will never occur if the system default value of the per-process file size limit is used; see `ulimit(2)` and `ULIMIT` in `<limits.h>`. This file size limit is expressed in 512-byte blocks; in terms of bytes, the default limit is greater than `ULONG_MAX`, thus `ULONG_MAX` will be exceeded before the file size limit is reached. A process may change its file size limit using `ulimit(2)`.

The error `EIO` will be reported if a physical hardware error occurs.

If `errno` has the value `EINTR` following a `write()`, no data was written.

6.5 Control operations on files

6.5.2 File control

Function: `fcntl()`

6.5.2.2 Description

Other than those defined, status bits are ignored if they are set when `fcntl()` is called with `F_SETFL` as the value for *cmd*.

A/UX supports advisory record locking for regular files only.

If `l_len` is negative when attempting to lock, the lock will succeed. However, this is not recommended as the checks for existing locks will not find conflicts if there are locks that were specified in this manner.

6.5.2.4 Errors

This implementation detects the conditions that would set `errno` to `EDEADLK`.

6.5.3 Reposition `read/write` file offset

6.5.3.2 Description

In the A/UX implementation, an `lseek()` on a device that is incapable of seeking will not report an error, and the file offset will be adjusted as if the call were successful.

7. Device-specific and class-specific functions

7.1 General terminal interface

A/UX runs on the Apple Macintosh family of computers. These systems all support two on-board, asynchronous communication ports. Usually referred to as the "modem" and "printer" ports, these ports may be accessed from the file system as `/dev/tty0` and `/dev/tty1`, respectively.

By default, the modem port supports modem control; however, the printer port does not. To enable modem control on the printer port, issue the following command; see `stty`(1) for more information:

```
stty -n /dev/tty1 modem
```

Modem control will remain enabled until the system is restarted; add the above command to the system startup script, `/etc/rc`, to reset the printer port each time the system is booted.

Additional communication ports may be added to Macintosh platforms using the system expansion slot(s). The hardware supplier must provide A/UX software drivers that support POSIX.1 semantics, including modem control and break transmission. For more information, see the A/UX Device Driver Kit, which is available from the Apple Programmers and Developers Association (APDA).

The Macintosh IIfx computer and Macintosh Quadra 900 computer support a serial input/output processor (IOP) that speeds data transfers for the on-board communication ports. The serial IOP does not fully support POSIX.1 semantics, however; see *Macintosh Technical Note #184* for more details. To enable complete POSIX.1 compatibility on these terminal interfaces, obtain the "serial switch" Control Panel Device (CDEV) from your Apple representative.

In addition to supporting asynchronous communications ports, the terminal interface supports network connections via the pseudo-terminal interface; see `pty`(4).

7.1.1 Interface characteristics

7.1.1.3 The controlling terminal

In this implementation, the controlling terminal for a session is allocated by the session leader when the session leader opens the first terminal device file that is not already associated with a session.

7.1.1.5 Input processing and reading

If `MAX_INPUT` is exceeded, the input queue is flushed.

7.1.1.6 Canonical mode input processing

If `MAX_CANON` is exceeded, the additional characters are discarded.

7.1.1.7 Noncanonical mode input processing

`MIN` is stored in an unsigned character; therefore, the value of `MIN` cannot be greater than 256, which is the value of `MAX_INPUT`.

7.1.1.8 Writing data and output processing

The `STREAMS` subsystem includes a buffering mechanism, however, the older, `clist` subsystem does not.

7.1.1.9 Special characters

The `START` and `STOP` characters cannot be changed. There are no multi-byte special character sequences. There are two additional single-byte special characters:

`EOL`	`ASCII NUL`	additional line delimiter
`NL`	`ASCII LF`	line delimiter

7.1.2 Parameters that can be set

7.1.2.1 `termios` structure

There is one additional member, `c_line`, in the `termios` structure in `<termios.h>`:

```
#define      NCCS         12

struct       termios {
             tcflag_t     c_iflag;
             tcflag_t     c_oflag;
             tcflag_t     c_cflag;
```

```
            tcflag_t    c_lflag;
            char        c_line;
            cc_t        c_cc[NCCS];
};
```

The `c_line` field specifies the line discipline number.

7.1.2.2 Input modes

The break condition is only defined for asynchronous data transmission. If the terminal supports other modes of transmission, the break condition is driver-specific.

`START` will be transmitted when the input queue is nearly empty. `STOP` will be transmitted when the input queue is nearly full.

The initial input control value is all bits clear.

7.1.2.3 Output modes

If `OPOST` is set, output characters are postprocessed as indicated by the remaining flags. Additional flags supported for `c_oflag` are:

`OLCUC`	map lowercase to uppercase on output
`ONLCR`	map NL to CR-NL on output
`OCRNL`	map CR to NL on output
`ONOCR`	no CR output at column zero
`ONLRET`	NL performs CR function
`OFILL`	use fill characters for delay
`OFDEL`	fill is DEL, else NUL
`NLDLY`	select newline delays
`TAB3`	
`BSDLY`	select backspace delays
`BS0`	
`BS1`	
`VTDLY`	select vertical-tab delay
`VT0`	
`VT1`	
`FFDLY`	select form feed delays
`FF0`	
`FF1`	

The initial output control value is all bits clear.

7.1.2.4 Control modes

The control modes are only defined for asynchronous data transmission. The initial hardware control values are `B9600`, `CS8`, and `CREAD`.

7.1.2.5 Local modes

If `ECHOE` and `ICANON` are set, the `ERASE` character will cause nothing to occur if there is no character to erase on the current line of the display.

If `ECHOK` and `ICANON` are set, the `KILL` character will cause this implementation to echo the newline character, `\n`, after the `KILL` character.

There are no implementation-defined functions associated with the input stream when `IEXTEN` is set. Setting `IEXTEN` has no effect on `ICANON`, `ISIG`, `IXON`, or `IXOFF`.

The initial local control value is all bits clear.

7.1.2.6 Special control characters

The number of elements in the `c_cc` array is the value of `NCCS`. `NCCS` is currently defined in `<termios.h>` to be 12.

A/UX does not support changing the `START` and `STOP` characters. The character values in the `c_cc` array indexed by the `VSTART` and `VSTOP` subscripts are ignored when `tcsetattr` is called.

The initial values of the control characters are described below:

`ESC`	`ASCII ESC`
`INTR`	`CONTROL-C`
`QUIT`	`ASCII FS`
`ERASE`	`DEL`
`KILL`	`CONTROL-U`
`EOF`	`CONTROL-D`
`START`	`CONTROL-Q`
`STOP`	`CONTROL-S`
`SWTCH`	`CONTROL-Z`
`SUSP`	`_POSIX_V_DISABLE`
`DSUSP`	`_POSIX_V_DISABLE`

7.1.3 Baud rate functions

Functions: `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`

7.1.3.2 Description

Attempts to set unsupported baud rates using these functions will not return an error.

7.1.3.4 Errors

A/UX will detect an error if the functions `cfsetispeed()` and `cfsetospeed()` are called with an invalid value for speed; the functions will return -1 and set `errno` to `EINVAL`.

7.2 General terminal interface control functions

7.2.2 Line control functions

Functions: `tcdrain()`, `tcflow()`, `tcflush()`, `tcsendbreak()`

7.2.2.2 Description

If *duration* is nonzero, `tcsendbreak` will not send a break.

If the terminal is not using asynchronous serial data transmission, the break condition is driver-specific.

8. Language-specific services for the C programming language

8.1 Referenced C language routines

A/UX conforms to the C Language Binding (Common-Usage C Language-Dependent System Support).

8.1.1 Extensions to time functions

If the first character of the environment variable `TZ` is a colon (`:`), the characters following the colon are interpreted as a pathname.

8.1.2 Extensions to `setlocale()` function

8.1.2.2 Description

This implementation supports only the "C" and "POSIX" locales.

If no non-NULL environment variable is present to supply a value, `setlocale` sets the specified category to the "C" locale.

8.2 C Language input/output functions

8.2.2 Open a stream on a file descriptor

Function: `fdopen()`

8.2.2.2 Description

There are no addition values for the *type* argument for `fdopen()`.

8.3 Other C language functions

8.3.2 Set time zone

Function: `tzset()`

8.3.2.2 Description

If `TZ` is absent from the environment, the default time-zone information is determined from the file `/etc/zoneinfo/localtime`; see `tzfile`(4).

9. System databases

9.1 System databases

The group and user databases are implemented by the files `/etc/group` and `/etc/passwd`, respectively.

If the initial user program field in the user database is null, the Bourne command shell will be used; see `sh`(1).

If the initial working directory field in the user database is null, the user's home directory will be the root directory.

There are two additional fields in a password file entry. An encrypted password field follows the login name and a field for the user's real name follows the numeric group id. There is an optional comment field that follows the numeric gid field.

A group file entry has an encrypted password field following the name field.

9.2 Database access

9.2.1 Group database access

Functions: `getgrgid()`, `getgrnam()`

9.2.1.3 Returns

The functions `getgrgid()` and `getgrnam()` access the same static data, thus the result of a call to either function may be overwritten by a subsequent call or a call to the other routine.

9.2.2 User database access

Functions: `getpwuid()`, `getpwnam()`

9.2.2.3 Returns

The functions `getpwuid()` and `getpwnam()` access the same static data, thus the result of a call to either function may be overwritten by a subsequent call or a call to the other routine.

This `passwd` structure is defined in `<pwd.h>`:

```
struct        passwd {

              char         *pw_name;
              char         *pw_passwd;
              uid_t        *pw_uid;
              gid_t        *pw_gid;
              char         *pw_age;
              char         *pw_comment;
              char         *pw_gecos;
              char         *pw_dir;
              char         *pw_shell;
};
```

This group structure is defined in `<grp.h>`:

```
struct        group {
              char          *gr_name;
              char          *gr_passwd;
              gid_t         gr_gid;
              char          **gr_mem;
};
```

10. Data interchange format

10.1 Archive/interchange file format

`pax`(1) may be used to read and create archives.

10.1.1 Extended `tar` format

A/UX supports the use of characters outside the portable filename characters set in names for files, users, and groups. The Macintosh character set is defined in *Inside Macintosh,* Volume I, page 247.

If a filename is found on the medium that would create an invalid pathname, the file will not be created and the data will not be stored.

This implementation allows the use of the `TSVTX` mode.

The *devmajor* and *devminor* fields are used to construct the value of `st_rdev` for device files created on the file system.

The value `S` is used in the *typeflag* field to represent the socket file type.

10.1.2 Extended `cpio` format

10.1.2.1 `cpio` header

The values of `c_dev`, `c_ino`, and `c_rdev` are the values in the corresponding fields of the data structure returned by `stat()`.

Special files are created with the major and minor numbers specified by `st_rdev` for the file in the archive.

10.1.2.2 `cpio` filename

If a filename is found on the medium that would create an invalid pathname, the file will not be created and the data will not be stored.

10.1.2.4 `cpio` special entries

For other special files, `c_filesize` is set to zero.

10.1.2.5 `cpio` values

`C_ISVTX`, `C_ISLNK`, and `C_ISSOCK` are supported in this implementation.

10.1.3 Multiple volumes

The user will be prompted for the next file when `EOF` is encountered.

Index

The Apple Publishing System

A/UX Programming Languages and Tools, Volume 1, was written, edited, and composed on a desktop publishing system using Apple Macintosh computers, an AppleTalk network system, Microsoft Word, and QuarkXPress. Line art was created with Adobe Illustrator. Proof pages were printed on Apple LaserWriter printers. Final pages were output directly to 70-mm film on an Electrocomp 2000 Electron Beam Recorder. PostScript®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated.

Text and display type are Apple's corporate font, a condensed version of ITC Garamond®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.

Writer: J. Eric Akin
Developmental Editors: Paul Dreyfus and Scott Smith
Design Director: Lisa Mirski
Art Director: Tamara Whiteside
Production Editor: Debbie McDaniel

Special thanks to Jeannette Allen, Tom Berry, Kristi Fredrickson, Gene Garbutt, Michael Hinkson, John Morley, John Sovereign, Kristen Webster, Laura Wirth, and Chris Wozniak