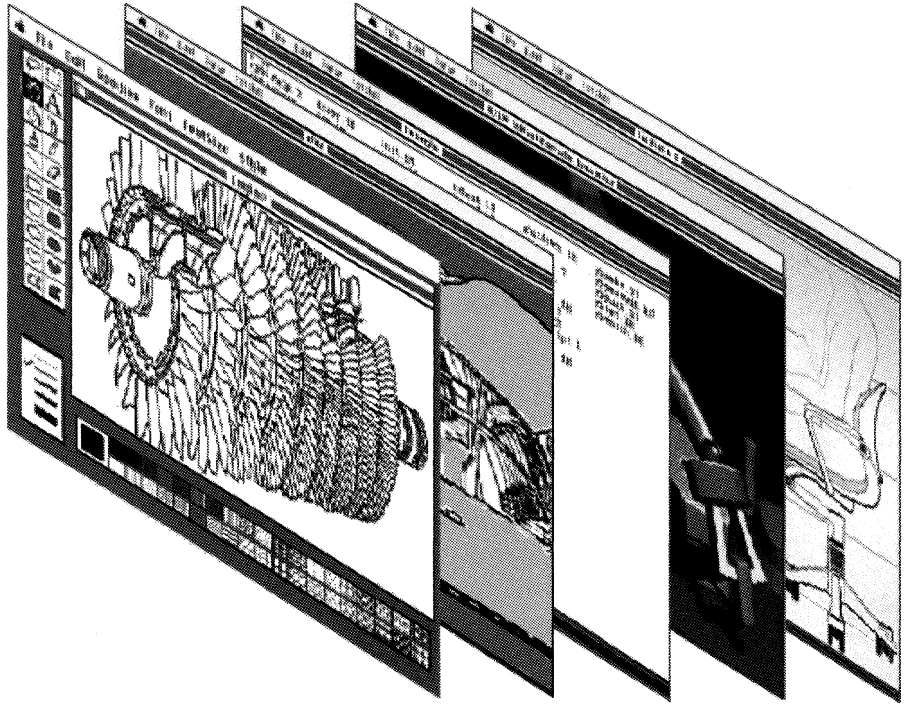
Apple® A/UX™ Programming Languages and Tools, Volume 1



Copyright

Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010

Limited Warranty on Media an Replacement

If you discover physical defects in the mar als distributed with an Apple product or in media on which a software product is distr buted, Apple will replace the media or mai als at no charge to you, provided you retur the item to be replaced with proof of purch to Apple or an authorized Apple dealer dui the 90-day period after you purchased the software. In addition, Apple will replace d aged software media and manuals for as lo as the software product is included in Appl Media Exchange Program. While not an upgrade or update method, this program offers additional protection for up to two years or more from the date of your origin purchase. See your authorized Apple deale for program coverage and details. In some countries the replacement period may be d ferent; check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON TH MEDIA AND MANUALS, INCLUDIN(IMPLIED WARRANTIES OF MER-CHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE, ARE LIM-ITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE OR GINAL RETAIL PURCHASE OF THI PRODUCT.

Even though Apple has tested the software and reviewed the documentation, **APPLE AND ITS SOFTWARE SUPPLIER MA NO WARRANTIES OR**

REPRESENTATIONS, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD AS IS, AND YOU THE PURCHASER ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.

IN NO EVENT WILL APPLE OR ITS SOFTWARE SUPPLIER BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION, even if advised of the possibility of such damages. In particular, Apple and its software supplier shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering such programs or data.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

A/UX Programming Languages and Tools, Volume 1

Contents

Preface

Conventions Used in This Manual

Throughout the A/UX manuals, words that must be typed exactly as shown or that would actually appear on the screen are in `Courier` type. Words that you must replace with actual values appear in *italics* (for example, *user-name* might have an actual value of `joe`). Key names appear in CAPS (for example, RETURN). Special terms are in **bold** type when they are introduced; many of these terms are also defined in the glossary in the *A/UX System Overview*.

Syntax notation

All A/UX manuals use the following conventions to represent command syntax. A typical A/UX command has the form

`command` [*flag-option*] [*argument*]...

where:

`command`	Command name (the name of an executable file).
flag-option	One or more flag options. Historically, flag options have the form -[*opt*...] where *opt* is a letter representing an option. The form of flag options varies from program to program. Note that with respect to flag options, the notation `[-a][-b][-c]` means you can select one or more letters from the list enclosed in brackets. If you select more than one letter you use only one hyphen, for example, `-ab`.
argument	Represents an argument to the command, in this context usually a filename or symbols representing one or more filenames.
[]	Surround an optional item.

...	Follows an argument that may be repeated any number of times.
`Courier` type	anywhere in the syntax diagram indicates that characters must be typed literally as shown.
italics	for an argument name indicates that a value must be supplied for that argument.

Other conventions used in this manual are:

<CR>	indicates that the RETURN key must be pressed.
^*x*	An abbreviation for CONTROL-*x*, where *x* may be any key.
cmd(*sect*)	A cross-reference to an A/UX reference manual. *cmd* is the name of a command, program, or other facility, and *sect* is the section number where the entry resides. For example, `cat`(1).

Chapter 1
Overview of the A/UX Programming Environment

Contents

Tables

Chapter 1

Overview of the A/UX Programming Environment

1. Introduction

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

There are four main kinds of tools that you will 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`) may 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*. We assume that the reader is conversant with the C programming language and with the general process of coding, compiling, testing, debugging, and so forth.

2. 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.

In very many 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, indeed, 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.

Various aspects of the C language are covered in detail in the next six chapters. The Fortran language, in its various A/UX incarnations, is discussed in Chapters 10 through 12.

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

`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 have compiled your program using the `-g` option. `sdb` is discussed in detail in Chapter 9.

The linker and assembler used automatically by the compilers are

`ld` The link editor

`as` The A/UX assembler for the Motorola 68020

Chapter 13 provides a complete reference manual for `as`. For a technical discussion of `ld`, see Chapter 14.

3. 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.

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 four main C language libraries in the A/UX programming environment are

`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.

`libm` This is the mathematical library for C language programs. This library provides exponential, Bessel functions, logarithmic, hyperbolic, and trigonometric functions. It is covered in detail in Chapter 6.

`libld` This library provides functions for the access and manipulation of common object files. It is covered in detail in Chapter 7.

`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*.

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

`libF77` This is the standard Fortran library. It includes various mathematical routines, string functions, and data conversion routines.

`libI77` This is the Fortran input/output library.

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.

4. The A/UX file system

4.1 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 do simple file manipulation. Programs can process data streams a 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 may be attached anywhere (possibly in multiple locations) on 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.

A file system breaks the logical device into four self-identifying regions:

1. The first block (address 0) is unused by the file system. It is left aside for booting procedures.
2. The second block (address 1) contains the so-called **super-block.** This block contains, among other things, the size of the disk and the boundaries of the other regions.
3. Following the super-block is the **ilist**, a list of file definitions. Each file definition is a 64-byte structure, called an **inode.** The offset of a particular inode within the ilist is called its **inumber.**

The combination of device name (major and minor numbers) and inumbers uniquely names a particular file.

4. After the ilist, and at the end of the disk, are free storage blocks available for the contents of files. The free space on a disk is maintained by a linked list of available disk blocks.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is used exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an inumber.

4.2 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 which may 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`.

4.3 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`.

4.4 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`.

4.5 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 see if 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").

A character device performs I/O one byte at a time. Input and output for character devices are considerably easier than for block devices; I/O requests from the user are sent to the device driver virtually untouched, bypassing the complicated buffer caching of block input and output. Characters generated by a user program are placed into a character queue until some limit is reached; then the physical I/O is performed.

A fifo is a special file that is also referred to as a "named pipe." Fifos are discussed, along with pipes, in "Pipes and Fifos."

5. 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.

5.1 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) for details.

5.2 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) for details.

5.3 File I/O

The A/UX system includes a number of system calls and subroutines for performing low-level input and output. We have already mentioned the `open` and `close` system calls, which, respectively, open and close files accessible to programs. 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 moves 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 moves 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 may also be moved explicitly, without performing any actual input or output, with the system call `lseek`. The position in the file to which the file pointer is to be moved may 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 may be controlled with the `fcntl` system call. For example, parts of the file may be **locked** to prevent either reading and/or writing those parts of the file. The `fcntl` call may also be used to duplicate file descriptors.

5.4 Pipes and fifos

The A/UX operating system supports yet a further 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 system call `pipe`. Once created, a pipe may 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`. Unlike named pipes, unnamed pipes are unidirectional: data may flow in only one direction through them. See `pipe`(2) for details.

A fifo special file is also called a **named pipe**, as it 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 they were written to the file, unlike normal files. Unlike unnamed pipes, 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.

5.5 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.

5.6 Asynchronous I/O

Asynchronous I/O happens most of the time when the I/O is both buffered and block.

When it happens, `reads` may 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.

When block caching was defined earlier (see the paragraph in "Special Files" on block devices), mention was made of the array of system buffers in which a block device caches blocks of some file. In fact, there are parallel arrays of buffers maintained, consisting of input buffers and output buffers. The input buffers receive the results of `reads`, while the output buffers hold intended `writes`.

When a `read` is requested, the results are shown immediately, synchronously with the request. Thus `reads` do not appear asynchronous, but may be so. If the data sought already have been cached into an input buffer, there is no need to read the data from disk, as they already were 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.

`writes` to disk are forced when:

- 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 following relation holds:

Table 1-1. Buffer vs. Disk Access with Asynchronous I/O

Process	Buffer Access	Disk Access
`read`	Synchronous	Asynchronous
`write`	Synchronous	Asynchronous

6. Process control

6.1 Process creation and termination

Processes are created by the system primitive `fork`. The newly created process (**child**) is a copy of the original process (**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` are shared after the `fork`. The processes are informed of their parts in the relationship, allowing them to select their own (usually nonidentical) destiny. The parent may wait for the termination of any of its children. This is accomplished through the `wait` system call.

A process may `exec` a file through use of the `exec` system calls. 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 did the `exec` persists, but after the `exec` it is executing a different program. Files that were open before the `exec` remain open after it.

If an executing program (for example, the first pass of a compiler) wishes to overlay itself with another program (for example, the second pass) then the executing program simply `exec`s 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 `exec`s a second program, it should first `fork` a child process, have the child `exec` the second program, and have the parent wait for the child. This is analogous to a subroutine call in the executing program.

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

6.2 Signals

The execution of a process can be controlled externally to the process by the use of **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 may be sent by the operating system, by the user from the shell, or from another user program; this is accomplished using either the shell command `kill` or the system call `kill`. The program to which the signal is sent may choose one of three ways to respond. The program receiving the signal may ignore the signal, it may 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 signal `SIGKILL` cannot be ignored by the receiving process.

A typical signal-handling scenario is as follows: A process indicates that it will catch designated signals via the `signal` system call. A call to `signal` 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 has 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 would check the global flag at intervals and, finding that a signal had arrived, would perform the appropriate processing.

User programs that need to process signals should have a separate signal-catching subroutine which 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 different methods it will use to process signals. A process can interpret signals in accordance with the System V Interface Definition (SVID) or in accordance with the conventions of the Berkeley Software Distribution, Release 4.2 (4.2 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's signal handling is reliable 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 will be caught and queued until they can be processed. Using SVID-compatible signals, this is not always the case; in certain circumstances, a program will lose signals, possibly resulting in the premature termination of the program. For more details, see `set42sig`(3) and `setcompat`(2).

6.3 Interprocess communication

The type of interaction between independent processes provided by signals is of a rather limited kind. In order 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. What allows it to function as a means of interprocess communication is that it is stored in a memory location that is accessible to various programs through certain system calls. By reading the values of semaphores and, possibly, 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 may 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 B-Net 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 the implementation of sockets and programming with them, see *A/UX Network Applications Programming*.

6.4 Program pause and wakeup

There are several ways to suspend execution of a program 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 may also 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 has been suspended with the `pause` system call, it is typically awakened with the signal `SIGALRM`. 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*`)` will instruct the calling process's alarm clock to send the signal `SIGALRM` to the calling process after *n* seconds. This call does not itself suspend execution of the calling process.

6.5 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.

7. Memory management

7.1 Dynamic memory allocation

Managing the available core memory is an important task for an operating system (like A/UX) which 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 may also need to allocate additional memory to an executing process. The A/UX environment provides a number of system calls and library routines for managing a program's use of memory storage.

The primary memory allocation request is `malloc`. A successful call of the form `malloc(`*n*`)` will return a pointer to *n* bytes of free memory. Memory may be returned to the operating system by calling the routine `free`. Other available memory allocation routines are `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).

7.2 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 like data base 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 system call `shmget`. Other processes may then gain access to this segment of

memory, provided that they possess permissions specified at the time the segment was created. A process may attach itself to a shared segment of memory by invoking the system call `shmat` and detach itself from that segment by invoking the system call `shmdt`. A shared memory segment is removed by using the system call `shmctl`; 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).

8. 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 UID, GID, current directory, and so on. The program may read the strings it finds in the environment, and modify its subsequent actions according to the results it receives. A program may also 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 may read the environment by executing a call of the form `getenv (`*name*`)`. It may alter the environment it receives from the shell by executing a call of the form `putenv (`*string*`)`, where *string* is of the form listed above.

It is a general characteristic of the A/UX operating system that a process can change only 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 process that calls it and of all processes that that process may create. Changes made to the environment do not persist after that process has exited. For further information, refer to `putenv`(3C) and `environ`(5).

9. 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*`)` will result in the program passing *string* to an instance of `/bin/sh` for execution, exactly as if *string* had been 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 are 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 may 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.

10. Error handling

The C language interface to the A/UX operating system provides a general facility for detecting and reporting error conditions which may 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 will return a non-negative integer which is the file descriptor of the opened file.

An unsuccessful system call returns a value of –1. In order 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:

1. a return value of –1 is returned to the calling program; and
2. 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 may 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 message printed 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 variable `errno`.

It is the responsibility of the calling program to detect and react to error conditions indicated by unsuccessful function calls. In addition to the variable `errno` and the subroutine `perror`, 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.

11. 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,* Volumes 1 through 5.

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*.

12. 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* will provide an overview of these various packages.

13. 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

`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 will 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 may be stored without using a lot of disk space, previous versions may be reconstructed easily, and versions can be kept track of with a simple, consistent numbering scheme.

`awk` — The `awk` programming language 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.

`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.

`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 has been 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.

`bc` — `bc` is a specialized language and compiler for handling arbitrary precision arithmetic using the `dc` calculator program.

`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.

`m4` `m4` is a general-purpose macro processor. The primary function of `m4` is to allow the replacement of some text by some (other) text. See also the standard C preprocessor (`cpp`).

`curses` The `curses` and `terminfo` packages provide a complete set of utility routines for writing screen-oriented programs.

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*.

In closing this overview, we should mention that the A/UX shells are themselves fully programmable interpreted languages. Shell scripts, therefore, can sometimes provide very rapid prototyping of programming tasks. As was 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's tools: pipes, input/output redirection, variables, quotation, and filename substitution. In very many instances, indeed, these shell scripts can serve as final versions of your program. The shell programming facilities are fully documented in *A/UX User Interface*.

Chapter 2
cc Command Syntax

Contents

Chapter 2

cc Command Syntax

1. Using cc

The cc command is a front-end program that invokes the preprocessor, compiler, assembler, and linkage editor, as appropriate. (The default is to invoke each one in turn.)

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

1.1 Command syntax

The syntax for cc is

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

where *flagopt* is zero or more flag options (see "Options") 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.

1.2 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 directiory; 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 linkage editor. The sections that follow describe these flag options.

2. Options

All options recognized by the `cc` command are listed below.

2.1 Recognized and executed by `cc`

Option	Argument	Description
`-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. To link a program that does not have floating-point code, the libraries `-lcno881` and `-lmno881` must be included on the command line.
`-fm68881`	none	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 which is linked in such a manner that the text can be made read-only and shared (nonvirtual) or paged (virtual).

`-p`	none	Reserved for invoking a profiler.
`-S`	none	Compile the named C programs, and leave the assembler-language output within corresponding files suffixed `.s`.
`-t`	`[p012al]`	Find only the designated preprocessor (`p`), compiler (`0` and `1`), 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`. The value of `-t ""` is equivalent to `-tp012`.
`-B`	*string*	Construct pathnames for substitute preprocessor, compiler, and link editor passes by concatenating *string* with the suffixes `cpp`, `c0` (or `ccom` or `comp`), `c1`, `c2` (or `optim`), `as` and `ld`. If *string* is empty it is taken to be `/lib/o`.
`-E`	none	Same as the `-P` option except output is directed to the standard output.
`-O`	none	Invoke an object code optimizer.
`-P`	none	Suppress compilation and loading; that is, invoke only the preprocessor and leave the output on corresponding files with the extension `.i`.

`-R`	none	Have assembler remove its input file when done.
`-T`	none	Truncate symbol names to 8 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 [`p012al`], 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 68020.
`-Z`	*flags*	Special *flags* to override the default behavior (see `cc`(1)). Currently recognized *flags* are:

`c` suppress returning pointers in both a0 and 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 inline code for MC68881 floating point coprocessor

`l` suppress selection of a loader command file

		`t` do not delete temporary files `F` flip byte order of multicharacter character constants
`-#`	none	Special debug option which, without actually starting the program, echoes the names and arguments of subprocesses which would have started.

2.2 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.
`-o`	*outfile*	Same as `-o` in `ld`(1). Produce an output object file, *outfile*. The default name of the object file is `a.out`.
`-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.

2.3 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.

By using appropriate options, you can terminate compilation early to produce one of several intermediate translations. For example,

`-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 root 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 file name is changed to one with the extension `.c`. Except for those produced by the preprocessor, any intermediate files may be saved and resubmitted to the `cc` command, with other files or libraries included as necessary.

`-W` This option lets you specify options for each step that is normally invoked from the `cc` command line, that is, (1) preprocessing, (2) the first pass of the compiler, (3) the second pass of the compiler, (4) optimization, (5) assembly, and (6) link editing.

At this time, only assembler and link editor options can be used with the `-W` option. The most common example of the `-W` option is

`-Wl,-VS,`*n*

which passes the `-VS`*n* option to the link editor (`ld`(1)). In the following example,

`-Wa,`*-option*

the compiler will pass the *-option* to the assembler.

`-O` This option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. When the optimizer is used, line numbers used for symbolic debugging may be transposed.

`-g` This option produces information for a symbolic debugger. (For more information see Chapter 9, "`sdb` Reference.")

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

Chapter 3
C Language Reference

Contents

Tables

Chapter 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.

1. Notation conventions

In the syntax notation used in this chapter, syntactic categories are indicated by *italic* type, and literal words and characters in `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."

2. 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 has been 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.

2.1 Comments

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

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

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

2.2 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 will accept identifiers up to 1024 characters long. Other implementations truncate identifiers to 7 or 8 characters, so long identifier names are not recommended.

2.3 Keywords

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

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

2.4 Constants

There are several kinds of constants, each of which has a type. The introduction to types is given in the "Names" section. Hardware characteristics that affect sizes are summarized in the subsection "Hardware Characteristics" under the general heading "Lexical Conventions." See also Chapter 4, "C Implementation Notes."

2.4.1 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 `0x` 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 hex constant that exceeds the largest unsigned machine integer is likewise taken to be `long`. Otherwise, integer constants are `int`.

2.4.2 Explicit long constants

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

2.4.3 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's character set.

Multicharacter character constants are permitted on the 68020. Multicharacter character constants can be told 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 backward. The `-ZF` flag option reverses the order of character assignment within the word. For example, when you compile a program including the line

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

`i` is assigned the value `0x64636261`, corresponding to `'dcba'`. If you compile the same program with the `-ZF` flag option, `i` is assigned the value `0x61626364`, corresponding to `'abcd'`.

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

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`.

2.4.4 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 may be missing, but not both. Either the decimal point or the `e` and exponent may be missing, but not both. Every floating constant has type `double`.

2.4.5 Enumeration constants

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

2.5 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 may be used.

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

2.6 Hardware characteristics

The following table summarizes certain hardware properties for the 68020. Note that the ranges for `float` and `double` are approximate.

Table 3-2. 68020 hardware characteristics

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

For more information on 68020 data representation, see Chapter 4, "C Implementation Notes."

3. 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 identifier's storage.

3.1 Storage class

There are four declarable storage classes:

- **Automatic variables** are local to each invocation of a block and are discarded upon exit from the block.
- **Static variables** are local to a block but retain their values upon reentry to a block even after control has left the block.

- **External variables** exist and retain their values throughout the execution of the entire program. They may 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 on exit from the block.

3.2 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's 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 may be stored into character variables, but the implementation is machine dependent. In particular, `char` may 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 may 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" for the sizes of types on the 68020.)

`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 usefully be interpreted as numbers, they are referred to as **arithmetic types.** `char`, `int` of all sizes whether `unsigned` or not, and `enum` are collectively called **integral types.** The `float` and `double` types are collectively called **floating types.**

The following table summarizes the categorization of fundamental types:

Table 3-3. Categorization of fundamental types

Type	Category		
	arithmetic	integral	floating
`char`	×	×	
`double`	×		×
`enum`		×	
`float`	×		×
`int`	×	×	
`long`	×	×	
`short`	×	×	

Besides 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.

4. Objects and lvalues

An **object** is a manipulatable region of storage. 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 below indicates whether it expects lvalue operands and whether it yields an lvalue.

5. Conversions

A number of operators may, 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."

5.1 Characters and Integers

A `char` or a `short` may 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 or not 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 may 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.

5.2 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`.

5.3 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 68020, negative floating values are rounded toward zero. The result is undefined if it will 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.

5.4 Pointers and integers

An expression of integral type may 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 (below). Two pointers to objects of the same type may be subtracted. In this case, the result is converted to an integer, as specified in the discussion of the subtraction operator (below).

5.5 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.

5.6 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 stays a `char`. If an `int` is the result of adding two `char`s, 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 is 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`, that is the type of the result.

6. 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." 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."

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 (`*`, `+`, `&`, `|`, `^`) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation, your program must use an explicit temporary.

The handling of overflow and divide check in expression evaluation is undefined. This implementation, like most that exist, ignores integer overflows. The integer division by 0 exception is enabled by default. The result of an integer division by 0 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.

6.1 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 below. Its declaration specifies its type. If the identifier's 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 may 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 above 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" under "Declarations").

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 under "Arrays, Pointers, and Subscripting" under "Types Revisited."

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 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" under "Declarations."

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 may 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 may 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."

6.2 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 (0), 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" 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" under "Declarations."

The `sizeof` operator yields its operand's 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`.

6.3 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 0. 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 0).

6.4 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 may 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 may 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 separating the objects pointed to. This conversion in general gives unexpected results unless the pointers point to objects in the same array, because pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

6.5 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 0 filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0 fill) if `E1` is `unsigned`; otherwise, it may be arithmetic.

6.6 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 0 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.

6.7 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 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be "null."

6.8 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.

6.9 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.

6.10 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.

6.11 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 0. Unlike &, && guarantees left-to-right evaluation. Moreover, the second operand is not evaluated if the first operand is 0.

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`.

6.12 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 0. 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`.

6.13 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 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

6.14 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 has taken 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 0 may be assigned to a pointer, however; it is guaranteed that this value will produce 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 may be a pointer, in which case the (integral) right operand is converted as explained in "Additive Operators." All right operands and all nonpointer left

operands must have arithmetic type.

6.15 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") and lists of initializers (see "Initialization" under "Declarations"), the comma operator as described in this subpart 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.

7. 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 described below.

7.1 Storage class specifiers

The storage class specifiers are

```
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" for more information). The meanings of the various storage classes are discussed in "Names."

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") 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 will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be 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.

7.2 Type specifiers

The type specifiers are

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` may be specified in conjunction with `int`; the meaning is the same as if `int` were not mentioned. The word `long` may be specified in conjunction with `float`; the meaning is the same as `double`. `unsigned` may 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 may 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 in "Structure and Union Declarations" and "Enumeration Declarations." Declarations with `typedef` names are discussed in "Typedef."

7.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

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

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in "Initialization." 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.

7.3.1 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 may be altered by parentheses (see the examples below).

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"). When several *array of* specifications are adjacent, a multidimensional array is created. The constant expressions that specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also 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 may 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 the possibilities of the above syntax are actually permitted. The restrictions are as follows: Functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions; likewise, a structure or union may not contain a function, but it may 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 (via indirection through a pointer). When this function is called, it returns an integer.

As another 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 3×5×7. 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]
```

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

7.4 Structure and union declarations

A structure is an object made up of a sequence of named members. Each member may 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 may also 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 may 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 may 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 0 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 but integer fields. Moreover, even `int` fields can be considered to be unsigned.

It is strongly recommended that fields be declared 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 0 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 may 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 may contain a pointer to an instance of itself.

You may use the third form of a structure or union specifier before a declaration that gives the specifier's complete specification 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 may not be used twice in the same structure, but the same name may 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 has been 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`.

7.5 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 may appear wherever constants are required. If no enumerators with = appear, the values of the corresponding constants begin at 0 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}`.

7.6 Initialization

A declarator may 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") or expressions that reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may 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 may not initialize unions or automatic aggregates.

You may, 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 will report an error if there are more initializers than members. If, however, the initializer does not begin with a left brace, only enough 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 0. Precisely the same effect could have been 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 0.

A further leap allows for the syntax of character array initialization. Because commas are common elements within strings, it would be 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.

7.7 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 would appear 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

7.8 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." 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 that could be specified in another way. Thus in the example above, `distance` is considered to have exactly the same type as any other `int` object.

8. Statements

Except as indicated, statements are executed in sequence.

8.1 Expression statement

Most statements are expression statements, which have the form

expression;

Usually expression statements are assignments or function calls.

8.2 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 bad 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.

8.3 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 0, the second substatement is executed. The "else" ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

8.4 `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.

8.5 `do` statement

The `do` statement has the form

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

The substatement is executed repeatedly until the value of the expression is 0. The test takes place after each execution of the statement.

8.6 `for` statement

The `for` statement has the form

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

This statement is equivalent to

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

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

The first expression specifies initialization for the loop; the second specifies a test made before each iteration such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementation that is performed after each iteration.

Any or all of the expressions may 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.

8.7 `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 may 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."

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's 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 "Break Statement."

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

8.8 `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.

8.9 `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 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").

8.10 `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* may be enclosed in parentheses.

8.11 `goto` statement

Control may be transferred unconditionally by means of the statement

```
goto identifier;
```

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

8.12 Labeled statement

Any statement may 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").

8.13 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`.

9. 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" in "Declarations") may also 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.

9.1 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" in "Scope Rules" 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 will be 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

9.2 External data definitions

An external data definition has the form

data-definition:
        declaration

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

10. Scope rules

A C program doesn't have to be compiled all 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 may 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.

10.1 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" in "Declarations"). 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 may 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 will be taken as a declaration with no declarators and with type `distance`.

10.2 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 identifiers' storage 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 will be passed on 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 will catch 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` via 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 will 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 may be declared `static`. This provides a way of hiding globals, and hence should be used with caution.

11. 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 may 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 may appear anywhere. Their effect lasts (independent of scope) until the end of the source program file.

11.1 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 may 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 may 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's preprocessor definition (if any) to be dropped.

If a `#defined` 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.

11.2 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). `#includes` may be nested.

11.3 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." Here, the restricted-constant expression cannot contain `sizeof` casts or an enumeration constant.)

A restricted-constant expression may also 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 have finished. During this evaluation, all identifiers undefined to the procedure evaluate to zero.

A control line of the form

`#ifdef` *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 `#ifdef (`*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 may 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 may be nested.

11.4 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.

12. 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`.

13. Types revisited

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

13.1 Structures and unions

Structures and unions may 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 `.` 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 may not be inspected unless that member had a value assigned more recently than any other member which overlaps the same space. One special guarantee is made by the language, however, in order 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) ...
```

13.2 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 could include

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

The definition of g might 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 (.

13.3 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 E2th 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 $i \times j \times \ldots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \ldots \times k$. 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 3×5 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.

13.4 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" under "Expressions" and "Type Names" under "Declarations."

A pointer may 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 may 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 may be converted to a pointer to another type. The resulting pointer may 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 may 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 `chars` have no alignment requirements; everything else must have an even address.

14. 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 operator

```
?:
```

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

When writing your program, you have more latitude with initializers. Besides constant expressions as discussed above, you can also use floating constants and arbitrary casts. You can also 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.

15. Portability considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant 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 proved not to be a problem. 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 can actually 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 character constants may 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" 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.

16. Syntax summary

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

16.1 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-list*$_{opt}$`)`
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.

16.2 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

16.3 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*
;

16.4 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*;

16.5 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*"

Chapter 4
C Implementation Notes

Contents

Figures

Chapter 4

C Implementation Notes

1. Introduction

This chapter describes the A/UX 68020 C programming language, including how data are represented, how data are 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 in order to match C code with the assembler. It is also intended for those who wish to write new system 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.

2. Data representations

In general, all data elements of whatever size are stored such 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) 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.

3. 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.

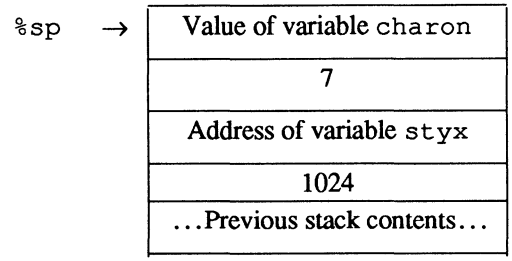
When a function is called, its parameters (if any) are evaluated and are then pushed onto the stack in reverse order. All parameters are pushed onto the stack as 32-bit `longs`, except for `floats` and `doubles`, which are pushed as 64-bit `doubles`. If a parameter is shorter than 32 bits, it is expanded 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:

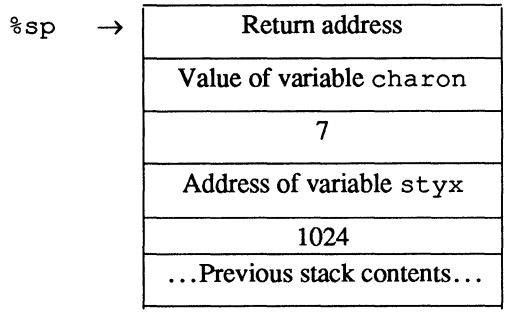
Figure 4-1. Stack contents after evaluation of function call



Functions are called by issuing either a `bsr` instruction or a `jsr` instruction, depending upon 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 upon the stack and then branches to the indicated function. After the call, on entry to the function, the stack looks like this:

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.

4. Setting up the stack

Upon entry into the function, the prolog code is executed. The **prolog code** allocates enough space on the stack for the local variables, plus enough 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.

5. Allocation of local variables and registers

A total of ten registers are available for register variables. Six of these are the 68020 data (`%d`) registers, and four are the 68020 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's 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` will fit 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 will be wrong. Use the `lint` program to find places where functions have not been declared (see Chapter 8, "`lint` Reference").

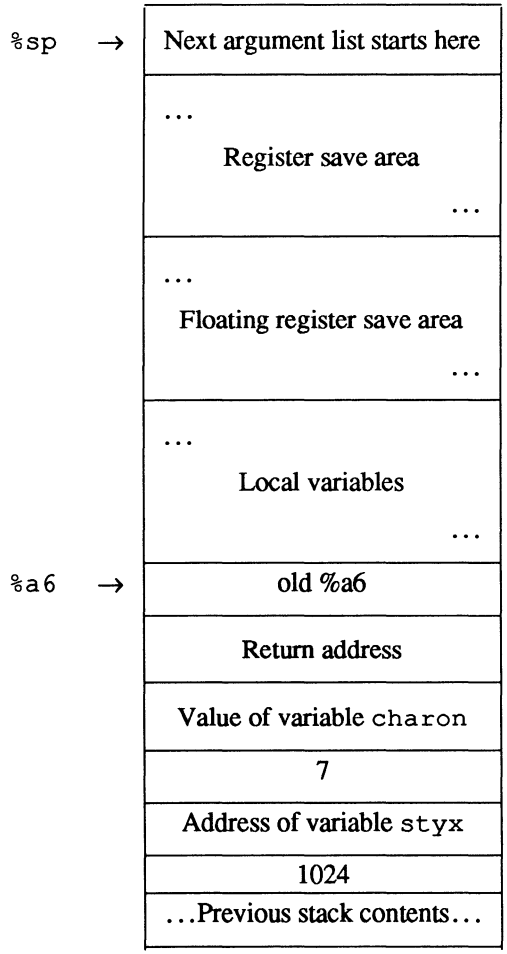
`pointer` register variables are assigned only to address registers, `float` and `double` register variables only to floating-point registers. Other register variables are assigned only to data registers. Register declarations are ignored for variables of type `struct` or `union`.

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 has completed, the stack looks like this:

Figure 4-3. Stack contents after execution of prolog code



6. 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 which 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 via 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.

7. 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.

8. Optimizations

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

`cc -O` *file*

generates optimized code.

9. 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 may 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.

10. Miscellaneous notes

The object files created by the assembler and linker use the common object file format (see Chapter 15, "COFF Reference").

The C compiler will accept 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.

Chapter 5
The Standard C Library (`libc`)

Contents

Chapter 5

The Standard C Library (`libc`)

1. Introduction

This chapter describes the A/UX C library. A **library** is a collection of related functions and/or declarations. Using a library simplifies programming effort 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 further on in this chapter.

2. 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 file 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

3. 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.

4. 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 need to 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

4.1 File access functions

Function	Reference	Brief description
`fclose`	`fclose`(3S)	Close an open stream.
`fdopen`	`fopen`(3S)	Associate stream with an `open`(2)ed file.

`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.
`vsetbuf`	`setbuf`(3S)	Similar to `setbuf`, but allowing finer control.

4.2 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.

4.3 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.

4.4 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(5) argument list.
vprintf	vprint(3C)	Print using format to standard output by varargs(5) argument list.
vsprintf	vprintf(3C)	Print using format to stream string by varargs(5) argument list.

4.5 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	tmpnam(3S)	Create temporary filename using directory and prefix.
tmpnam	tmpnam(3S)	Create temporary filename.
tmpfile	tmpfile(3S)	Create temporary file.

5. 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)	Copy 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.

6. 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>
```

6.1 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.

6.2 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.

7. Time functions

These functions are used for gaining access to and reformatting the system's idea of the current date and time. 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.

8. 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.

8.1 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.

8.2 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`.

8.3 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.

8.4 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 a specified file.

8.5 Parameter access

The following functions provide access to several different types of

parameters. None require 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 an environment variable.

8.6 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.

8.7 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.

8.8 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.

8.9 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.
`mallinfo`	`malloc`(3X)	Space usage.
`realloc`	`malloc`(3X)	Change size of allocated storage.

8.10 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 upon the linear congruent algorithm and 48-bit integer arithmetic. The `rand` and `srand` functions provide an interface to a multiplicative congruential random number generator with 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.
srand	rand(3C)	Seed the generator for rand.
srand48	drand48(3C)	Seed the generator for drand48, lrand48, and mranb48 using a long.

8.11 Signal handling functions

The functions gsignal and ssignal implement a software facility similar to signal(3) in *A/UX Command 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.

8.12 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.

Chapter 6
The C Math Library

Contents

Chapter 6

The C Math Library

1. Introduction

This chapter describes the A/UX math library. A **library** is a collection of related functions and/or 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*.

The math library is made up of functions and a header file. The functions may 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>
```

2. The math library functions

The math library functions are grouped into the following categories:

- Trigonometric functions
- Bessel functions
- Hyperbolic functions
- Miscellaneous functions

2.1 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.

2.2 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*.

2.3 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.

2.4 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.
`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.

Chapter 7
The C Object Library

Contents

Chapter 7

The C Object Library

1. Introduction

This chapter describes the A/UX object library. A **library** is a collection of related functions and/or declarations. 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 can be found in Section 5 of *A/UX Programmer's Reference*.

The object file library provides functions for the access and manipulation of 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 15, "COFF Reference" in this manual.

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

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

```
cc file -lld
```

This 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>
```

2. The object library functions

Function	Reference	Brief description
`ldaclose`	`ldclose`(3X)	Close object file being processed.
`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 the name of an object file symbol table entry.
`ldlinit`	`ldlread`(3X)	Prepare object file for reading line number entries via `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 being processed given the name of a section.

`ldnshread`	`ldshread(3X)`	Read section header of the named section of the object file being processed.
`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.
`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.

3. 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*.

Chapter 8
`lint` Reference

Contents

Chapter 8

`lint` Reference

1. `lint`: A C program checker

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 will accept 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 `lint`'s checking mechanisms if they aren't necessary for a given application.

2. Using `lint`

The `lint` command has the form

`lint` [*option* ...] *file* ... *library-descriptor* ...

where *option*s are optional flags that control `lint` checking and messages, *file*s 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-descriptor*s 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 which 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 will print messages if the source files contain a redefinition of a library routine.

2.1 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

`-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) may 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 might seem.)

`-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 wish 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

- 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 a program's 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 either used and not defined or defined and not used. For more information, please refer to "Unused Variables and Functions" later in this chapter.

`-v` Use this option to suppress messages concerning unused function arguments. For more information, please 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 language 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.

3. Message categories

The following subsections describe the major categories of messages printed by `lint`.

3.1 Unused variables and functions

As sets of programs evolve and develop, variables and function arguments that were used previously may fall into disuse. 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 to 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();
```

will evoke 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 number of arguments in calls to a function. If you wish 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 may not be used. Conversely, many functions and variables defined elsewhere may be used. The `-u` option may be used to suppress the spurious messages that might otherwise appear.

3.2 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 may 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 may also be symptomatic of bugs.

3.3 Flow of control

The `lint` program tries to detect unreachable portions of the programs that it processes. It will print 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 may 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` may cause unreachable code that `lint` does not detect. This can seriously affect the determination of returned function values (see "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 will inform `lint` that a portion of the program cannot be reached.

If you give the `-b` option, `lint` will not print a message about unreachable `break` statements. Programs generated by `yacc` and especially `lex` may 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 wish to get these messages, you can invoke `lint` without the `-b` option.

3.4 Function values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function "values" that have never been 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 will give 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` will call `g` and return with no defined return value. This will trigger a message from `lint`. If `g`, like `exit`, never returns, the message still will be 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 may constitute an inefficiency in the function definition. When the value is seldom used, it may 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.

3.5 Type checking

The `lint` program enforces the C language type-checking rules more strictly than the compilers do. 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 `.` 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 will prevent strict type checking for the next line in the program only.

3.6 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 has been 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.

3.7 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` will print messages about certain comparisons and assignments being illegal or nonportable. For example,

```
char c;
    ...
if ((c = getchar()) < 0)...
```

will work on one machine but will fail 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 bit fields. When constant values are assigned to bit fields, the field may be too small to hold the value. This is true especially because on some machines bit fields are considered signed quantities. While it may seem logical to consider that a two-bit field declared of type `int` cannot hold the value 3, the problem disappears if the bit field is declared to have type `unsigned`.

3.8 Assignments of `longs` to `ints`

Bugs may arise from the assignment of `long` to an `int`, which may truncate the contents. (Truncation happens only when `longs` hold a longer quantity than `ints`. In the current implementation, `longs` are the same length as `ints`.) This may happen in programs that have been incompletely converted to use `typedefs`. When a `typedef` variable is changed from `int` to `long`, the program may stop working. This is because some intermediate results may be assigned to `ints`, which are truncated. Because there are a number of legitimate reasons for assigning `longs` to `ints`, the detection of these assignments is disabled by the `-a` option. If `lint` is using the `-p` option to detect possible portability problems, however, it may print the message

```
warning: conversion from long may lose accuracy
```

even if you're using the `-a` option.

3.9 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 will never succeed. For a third example,

```
unsigned x;
if( x > 0)  ...
```

is equivalent to

```
if( x != 0)
```

which may not be the intended action. The `lint` program will print the message

```
degenerate unsigned comparison
```

in these latter two cases.

If a program contains something similar to

```
if( 1 != 0 )...
```

`lint` will print 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 exacerbated 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 enclose such expressions in parentheses; `lint` encourages this with an appropriate message.

When the `-h` option has not been 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.

3.10 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) could cause ambiguous expressions. For example,

```
a =-1;
```

could 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 the abandonment of 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 equals sign between the variable and the initializer. For example,

```
int x = -1;
```

This is free of any possible syntactic ambiguity.

3.11 Pointer alignment

Certain pointer assignments may 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. The message

```
possible pointer alignment problem
```

will appear.

3.12 Multiple uses and side effects

In complicated expressions, the best order in which to evaluate subexpressions may 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 may or may 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 will 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.

Chapter 9
sdb Reference

Contents

Figures

Chapter 9

sdb Reference

1. sdb: A symbolic debugger

This chapter describes the symbolic debugger sdb(1) as implemented for the C language and Fortran 77 compilers (cc and f77) on the A/UX operating system. The sdb program is useful both for examining core images of aborted programs and for providing an environment in which you can monitor and control the execution of a program.

The sdb program allows you to interact with a debugged program at the source language level. When debugging a core image from an aborted 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 may place breakpoints at selected statements or single step 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 may call procedures directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines that provide formatted printout of structured data.

2. 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 has been specified, you can use sdb to obtain a trace of the called functions at the time of the abort 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;
*
```

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` has a notion of current function and current line. In this example, they are initially set to `main` and `25`, respectively.

2.1 Arguments

In the above example, `sdb` was called with one argument, `prgm`. In general, `sdb` takes 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 will be named `a.out`. `sdb`, however, will not work on old `a.out` format files. Only COFF files may 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 to reside in a single directory. The default is the working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a function that was not compiled with the `-g` option. In this case, `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 will print an error message if `main` was not compiled with the `-g` option, but debugging can continue for those routines compiled with the `-g` option.

2.2 Example

The following is a typical example of `sdb` use. The first example, Figure 9-1, is the source file used to create the output file shown in Figure 9-2, an illustration of a session with `sdb`.

Figure 9-1. Sample **sdb** input file

```
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
```

Figure 9-2. Sample **sdb** session.

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 proc name and line number
`*r`	Run the program
`a.out`	sdb echoes command line executed
`Breakpoint at` `div2:9: j = i>>1;`	Execution stops just before line 9
`*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
`*div2(1)/`	Run 'div2' with other arguments
`0`	
`*div2(-2)/`	
`-1`	
`*div2(-3)/`	
`-2`	
`*q`	

2.3 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)     [prgm.c:96]
  main(argc=1,argv=0x7fffff54,
        envp=0x7fffff5c)[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.

2.4 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 and `?` to match any single character. Consider the following commands:

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with `x`, the second prints the values of all two-letter variables in function `sub` beginning

with `y`, and the last 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

- `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 may 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.

There are a number of format specifiers available:

- `a` Print characters, starting at the variable's 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, gaining access to arrays may be limited to one-dimensional arrays. Note that as a special case

```
*psym->/d
```

displays the location pointed to by `psym` in decimal.

You can also display core locations by specifying their absolute addresses. The command

```
*1024/
```

displays location 1024 in decimal. As in the C language, numbers may also 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 so that

```
*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 =, so

```
*i=
```

displays the address of `i`. Another feature whose usefulness will become apparent later is the command

```
*./
```

which redisplays the last variable typed.

3. Display and manipulation

The `sdb` program has been 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` has a notion of current file and current line within the file.

The `sdb` program also knows how the lines of a file are partitioned into functions, so it has a notion of current function. As noted elsewhere, the current function is used by a number of `sdb` commands.

3.1 Displaying the source file

There are four commands for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are

`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. Advances the current line by ten.

CONTROL-d Scrolls; 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.

3.2 Displaying another source file or function

The `e` command is used to display a different source file. Either of the forms

```
*e  function
*e  file.c
```

may be used. The first 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. Finally, an `e` command with no argument causes the current function and filename to be printed.

3.3 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. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression. The second command searches backward through the file for the same thing. The trailing slash character (`/`) and question mark (`?`) may be omitted from these commands. Regular expression matching is identical to that of `ed`(1).

The + and - commands may be used to move the current line forward or backward by a specified number of lines. Typing a newline advances the current line by one, and typing a number causes that line

to become the current line in the file. These commands may be combined with the display commands so that

```
*+15z
```

advances the current line by 15 and then prints 10 lines.

4. A controlled testing environment

One very useful feature of `sdb` is breakpoint debugging. After entering `sdb`, certain lines in the source program may 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 has not been compiled with the `-g` option, execution will proceed until a statement in a function compiled with the `-g` option is reached.

You can also have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the `-g` option.

4.1 Setting and deleting breakpoints

You can set breakpoints at any line in a function that contains executable code. The command format is

```
*12b
*proc:12b
*proc:b
*b
```

The first form 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. The second form sets a breakpoint at line 12 of function `proc`, and the third sets a breakpoint at the first line of `proc`. The last sets a breakpoint at the current line.

You can delete breakpoints with the commands

```
*12d
*proc:12d
*proc:d
```

In addition, if the command `d` is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a `y` or `d`, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a `B` command, and the `D` command deletes all breakpoints. It is sometimes desirable to have `sdb` automatically perform a sequence of commands at a breakpoint and then have execution continue. 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. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the `a` command, execution continues after the function name or source line is printed.

4.2 Running the program

The `r` command is used to begin 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 had been typed on the shell command line. If no arguments are specified, the arguments from the last execution of the program are used. To run a program

with no arguments, use the `R` command.

After the program is started, 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 execution of a stopped program. A line number may be specified, as in

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the `c` command finishes. There is also a `c` command that continues but passes the signal that stopped the program back to 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. This command is useful if you want to avoid executing a section of code that is known to be bad. You should not attempt to continue execution in a function other than the one in which the breakpoint is located.

The `s` command is used 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. It is often used when you're confident that the called function works correctly but you're interested in testing the calling routine.

The `i` command is used to run 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.

4.3 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. The second is intended for calling functions; it executes the function and prints the value that it returns. The value is printed in decimal format unless some other format is specified by `m`. Arguments to functions may 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.

5. 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 and you can place breakpoints at arbitrary addresses. You can also use the `sdb` program to display or modify the contents of the machine registers.

5.1 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 instruction. 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`.

5.2 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`.

5.3 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 may be a number, character constant, register, or the name of another variable. If the variable is of type `float` or `double`, it can also be a floating-point constant.

Chapter 10
`f77` Command Syntax

Contents

Chapter 10

f77 Command Syntax

1. Using f77

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.

If EFL (compiler) source files are given as arguments to the f77 command, they will be translated into Fortran before being presented to this Fortran compiler (see "efl Reference" in this volume).

The f77 command invokes the C compiler to translate C source files and the assembler to translate assembler source files.

Object files will be 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.

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).

-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.

`-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 name `a.out` (default).

The *file* argument to `f77` may 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.

2. Related utilities

These utilities are useful adjuncts to `f77`. Their special characteristics are described in the following table:

`efl`	Compiles a program written in Extended Fortran Language (EFL) into Fortran 77. See "`efl` Reference" in this volume 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.

Chapter 11
Fortran Language Reference

Contents

Chapter 11

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.

1. 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.

2. Language extensions

2.1 `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.

2.2 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`.

2.3 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 will issue 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.

2.4 Recursion

Procedures may 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.

2.5 Automatic storage

`static` and `automatic` are recognized keywords in this implementation, but not in ANSI Standard Fortran 77. These keywords may 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 may not appear in `equivalence`, `data`, or `save` statements.

2.6 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.

2.7 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 will be recognized only if they appear in lowercase.

2.8 `include` statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file `stuff`. `include` statements may be nested to a reasonable depth, currently ten.

2.9 Binary initialization constants

A `logical`, `real`, or `integer` variable may 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.

2.10 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 quote mark, you may embed the other within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an `integer` word boundary. Each character string constant appearing outside a `data` statement is followed by a null character to ease communication with C language routines.

2.11 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 may be used in place of character string constants and may also be used to initialize noncharacter variables in data statements.

2.12 Equivalence statements

This compiler permits single subscripts in `equivalence` statements under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

2.13 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.

2.14 Commas in formatted input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile. When you request a formatted read of noncharacter variables, commas may 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
```

will be read correctly.

2.15 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 will be of type `integer*2`. If the precision of an integer-valued intrinsic function cannot be determined by the generic function rules, the compiler will choose 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` will be deemed `short`. Note that these `short integer` and `logical` quantities do not obey the standard rules for storage association.

2.16 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*.

3. 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 exceptions to the standard involve the following:

1. Double-precision alignment
2. Dummy procedure arguments
3. `t` and `tl` formats

3.1 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 `equivalenced` 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.
2. Load that double-precision temporary.

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.

3.2 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.

3.3 `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 has already been processed.

This compiler's implementation uses "seeks." Therefore, if the standard output unit is not one that allows seeks, such as a terminal, the program is in error.

Benefits of the implementation chosen include the following:

- 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.

4. 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:

1. Procedure names
2. Data representation
3. Return values
4. Argument lists

4.1 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.

4.2 Data representations

The following is a table of 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.

4.3 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 could 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( )
```

4.4 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
```

5. File formats

5.1 File structure

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 is made up of records of the given length. In the special case that the record length is given as 1, 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 will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's 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 will write 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 will be that the single record you thought was written will be treated as more than one record when being read or backspaced over.

5.2 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 will not be 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 has been opened explicitly for `sequential` I/O is positioned initially. In fact, the I/O system attempts to position the file at the end. A `write` will append to the file and a `read` will result in an end-of-file indication. To position a file at its beginning, use a `rewind` statement. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

Chapter 12
EFL Reference

Contents

Figures

Tables

Chapter 12

EFL Reference

1. EFL: An extended Fortran language

This chapter is a reference for the EFL programming language. It describes the features and use of the language, and, although supplemented by the chapters on Fortran, can stand alone as an arbiter of the EFL language. To use this chapter, you should have some familiarity 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 placement of comment indicators and data in standard, specified columns, and will not compile properly if these columns are not used. In addition, EFL is a structured language, while standard Fortran uses `gotos` and `continue` statements. These and other Fortran restrictions are mentioned in sections such as "Continuation Conventions" and "Miscellaneous Output Control Options."

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.

In this chapter's examples and syntax specifications, a construct surrounded by double brackets represents a list of one or more of those

items, separated by commas. Thus, the notation

[[*item*]]

could refer to any of the following:

item
item, item
item, item, item

To increase the legibility of EFL programs, you may break some of the statement forms without an explicit continuation. A square (□) in the syntax represents a point where an end-of-line will be ignored.

1.1 `efl` command syntax

The A/UX `efl` command has the following syntax:

`efl` [`-w`] [`-#`] [`-C`] [*filename...*]

The flag options for `efl` are:

`-w` Suppresses warning messages

`-#` Suppresses comments in the generated program and the flag option

`-C` (on by default) Causes comments to be included in the generated program

An argument with an embedded = (equals sign) sets an `efl` flag option as if it had appeared in an `option` statement at the start of the program. Many options are described in the section "Compiler Options." A set of defaults for a particular target machine may 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.

2. Lexical form

2.1 Character set

The following characters are legal in an EFL program:

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	, ; : . + - * / = < > & ~ | $

Figure 12-1. Legal characters in EFL

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 (`!`) may be used in place of a tilde (~) as the logical unary operator "complement." Square brackets (`[` and `]`) may 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.

2.2 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.

2.3 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 may 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 may continue lines explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been 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.

2.4 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.

2.5 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.

2.6 `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 may 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. `includes` may be nested at least ten deep.

2.7 Identifiers

An **identifier** is a name used in an EFL program consisting of a letter or a letter followed by letters or digits. Figure 12-2 shows a list of the reserved words that have special meaning in EFL, and therefore should not be used as identifiers.

```
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
```

Figure 12-2. Reserved words in EFL

You should use these words only for the purposes described in this chapter.

2.8 Strings

A **character string** is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double-quote marks ("), and vice versa. You may not break a quoted string across a line boundary. Legal character strings include

```
'hello there'
"ain't misbehavin'"
```

2.9 Integer constants

An **integer constant** is a sequence of one or more digits.

```
0
57
123456
```

2.10 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

Figure 12-3. Forms for floating-point constants in EFL

2.11 Punctuation

You may use certain characters to group or to separate objects in the language, as follows:

Parentheses	( )
Braces	{ }
Comma	,
Semicolon	;
Colon	:
End-of-line	<CR>

Figure 12-4. Characters for grouping or separating in EFL

The end-of-line is a token (statement separator) if the line is nonblank or noncontinued.

2.12 Operators

The EFL operators are written as sequences of one or more nonalphanumeric characters, as shown in Figure 12-5.

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 × b)
/	divided by (a + b)
**	exponentiation (a^b)
<	is less than (a < b)
<=	is less than or equals (a ≤ b)
>	is greater than (a > b)
>=	is greater than or equals (a ≥ b)
==	equals (a ≡ b)
~=	does not equal (a ≠ b)
$	repetition (2$a = aa)
.	fp decimal point (a.*exp field*)
&&	logical and (a ∧ b)
||	logical or (a ∨ b)
&	and (a and b)
|	or (a or b)
=	assign equals (a "gets" b)
+=	assign plus (a = a + b)
-=	assign minus (a = a - b)
/=	assign divide (a = a + b)
*=	assign times (a = a × b)
**=	assign exp (a = a^b)
&&=	assign logical and (a = a ∧ b)
||=	assign logical or (a = a ∨ b)
&=	assign and (a = a and b)
|=	assign or (a = a or b)
->	*leftside = structure name*

Figure 12-5. EFL operators

where "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") in which some of the operators may be represented by a string consisting of a dot, an identifier, and another dot (for example, `.lt.`).

2.13 Macros

EFL has a simple macro substitution facility. You may 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*.

3. Program form

3.1 Files

A **file** is a sequence of lines and is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside a procedure affect the succeeding procedures on that file.

3.2 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).

3.3 Block scope

You may 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" under "Executable Statements" for further information on blocks).

You may 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` will be 7 throughout the program. If you want to include a subroutine at a deeper level and that subroutine needs `a` to equal `3`, you may redefine `a` for that subroutine. `a` will equal `3` in that subroutine only, however, because, as soon as the program leaves the subroutine, the definition set forth in level 0 will prevail.

A procedure illustrating block level scope might look like the code shown in Figure 12-6.

```
#   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
```

Figure 12-6. Procedure illustrating block level scope

3.4 Statements

Statements are of the following types:

```
option
include
define

procedure
end

declarative
executable
```

The `option` statement is described in "Compiler Options." The `include`, `define`, and `end` statements have been described previously; you may not 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.

3.5 Labels

An executable statement may have a label, which may 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:` in Figure 12-7.

```
        read(, x)
        if(x < 3) goto error
        ...
error:      fatal("bad input")
```

Figure 12-7. Example of a label

4. Data types and variables

EFL supports a small number of basic (scalar) types. You may define objects made up of variables of basic type (that is, **aggregates**) and then define other aggregates in terms of previously defined aggregates.

4.1 Basic types

The basic types are

`logical`	A `logical` quantity may take on the two values `true` and `false`.
`integer`	An `integer` may 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`s.
`long real`	A `long real` is a more precise approximation to a rational. `long real`s 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`s.
`character(`*n*`)`	A `character` quantity is a fixed-length string of *n* characters.

4.2 Constants

There is a notation for a constant of each basic type.

A `logical` may take on the 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 may be preceded by a plus or minus sign. The following are valid `real` constants:

```
17.3
-.4
7.9e-6    ( = 7.9 × 10^-6)
14e9      ( = 1.4 × 10^10)
```

The following are valid `long real` constants:

```
7.9d-6 ( = 7.9 × 10^-6)
5d3
```

A `character` constant is a quoted string. Consider, for example, the following:

```
"bad input"
"I'm real, not integer"
```

4.3 Variables

A **variable** is a quantity with a name and a location; at any particular time the variable may also 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's **storage class** is the association of its name and its location. A storage class may 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 may be either global or local.

1. The names of common areas are global. **Global variables** may 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.")

Floating-point variables are either of normal or long precision. **Normal precision** is 32 bits; **long precision** is 64 bits. You may state this attribute independently of the basic type.

4.4 Arrays

You may 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 may 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 may include negative numbers. Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

For example, the declared integer array

```
array (2, 10) chance
```

might have the elements

```
chance (3)
chance (2, 8)
```

4.5 Structures

You may 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 may 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 may be of any type (including previously defined structures), or they may be arrays of such objects. You may pass entire structures to procedures or use them in input/output lists; you may also 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
   }
```

5. Expressions

Expressions are syntactic forms that yield a value. An expression may 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."

Table 12-1. Precedence of operators in EFL

Operator	Meaning	Priority
->	*leftside = structure name*	Highest
.	fp decimal point (a.*exp field*)	
**	exponentiation (a^b)	
*	times (a × b)	
/	divided by (a + 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 equals (a ≤ b)	
>	is greater than (a > b)	
>=	is greater than or equals (a ≥ b)	
==	equals (a ≡ b)	
~=	does not equal (a ≠ b)	
&	and (a and b)	
&&	logical and (a ∧ b)	
|	or (a or b)	
||	logical or (a ∨ b)	
$	repetition (2$a = aa)	
=	assignment (a "gets" b)	Lowest
+=	assign plus (a = a + b)	
-=	assign minus (a = a - b)	
*=	assign times (a = a × b)	
/=	assign divide (a = a + b)	
**=	assign exp ($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
```

5.1 Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

5.1.1 Constants

Constants are described in "Constants" under "Data Types and Variables."

5.1.2 Variables

Scalar variable names are primaries. They may appear on the left or right side of an assignment. Unqualified names of aggregates (structures or arrays) may appear only as procedure arguments and in input/output lists.

5.1.3 Array elements

You may 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)
```

5.1.4 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 may be further qualified.

```
a.b
x(3).y(4).z(5)
```

5.1.5 Procedure invocations

You may 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" under "Declarations"), the name of a function known to the EFL compiler (see "Known Functions" under "Procedures"), 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 are examples of 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 may use the corresponding formal argument as the left side of an assignment or in an input list; otherwise, it may 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").

5.1.6 Input/output expressions

The EFL input/output syntactic forms may be used as integer primaries that have a nonzero value if an error occurs during the input or output.

5.1.7 Coercions

You may **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 may be coerced to any other arithmetic type. A character expression of one length may be coerced to a character expression of another length. Logical expressions may not be coerced to a nonlogical type.

As a special case, a quantity of `complex` or `long complex` type may 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.

5.1.8 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* may denote a variable, array, array element, or structure member. In the second case, *attributes* may 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 may 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 may not 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)
```

5.2 Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression will be evaluated before any larger expression of which it is a part is evaluated.

5.3 Unary operators

All the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

5.3.1 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.

5.3.2 Logical

The only logical unary operator is complement (~). This operator is defined by the equations

```
~ true = false
~ false = true
```

5.4 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.

5.4.1 Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative:

```
a**b**c = a**(b**c) = a^(b^c)
```

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 12-2.

Table 12-2. 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

where i = `integer`, r = `real`, c = `complex`, lr = `long real`, 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).

5.4.2 Logical

The two binary logical operations in EFL, `and` and `or`, are defined by the truth tables shown in Table 12-3.

Table 12-3. 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 may evaluate the operands in any order, thus speeding up the code.

5.5 Relational operators

There are six relations between arithmetic quantities. These operators are not associative.

Table 12-4. 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

Because the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~=. The character collating sequence is not defined.

5.6 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 there 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.

5.7 Dynamic structures

EFL does not have an address (pointer, reference) type. There is a notation, however, for dynamic structures:

leftside `->` *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template on the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* 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`.

5.8 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)
```

5.9 Constant expressions

If you build an expression out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

6. Declarations

Declarations statements describe the meaning, shape, and size of named objects in the EFL language.

6.1 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. Examples of declarations are shown in Figure 12-8.

```
integer k=2

long real b(7,3)

common(cname)
      {
      integer i
      long real array(5,0:3) x, y
      character(7) ch
      }
```

Figure 12-8. Examples of EFL declarations

6.2 Attributes

The following are basic types in declarations:

```
logical
integer
field(m:n)
character(k)
real
complex
```

Figure 12-9. Basic EFL types

In the above list, the quantities *k*, *m*, and *n* denote integer constant expressions with the properties $k > 0$ and $n > m$.

6.2.1 Arrays

The dimensionality can be declared by an `array` attribute:

`array(` $b_1,\ldots,b_n$ `)`

Each of the b_i may 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 will recognize important cases such as `(0:n-1)`.) The upper bound for the last dimension (b_n) may be marked by an asterisk (`*`) if the size of the array is unknown.

The following are legal `array` attributes:

```
array(5)
array(5, 1:5, -3:0)
array(5, *)
array(0:m-1, m)
```

Figure 12-10. Examples of legal array attributes

6.2.2 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 may appear as a member of any number of structures. It may also be the name of an ordinary variable, as a structure member name is used only in contexts where the parent type is known.

Figure 12-11 shows valid structure attributes.

```
struct xx
        {
        integer a, b
        real x(5)
        }
struct { xx z(3); character(5) y }
```

Figure 12-11. Examples of valid structure attributes

The last line defines a structure that contains an array of three `xxs` and a character string.

6.2.3 Precision

Variables of floating-point (`real` or `complex`) type may be declared to be `long` to ensure that they have higher precision than ordinary floating-point variables. The default precision is `short`.

6.2.4 Common

Certain objects called "common areas" have external scope, and may 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.

6.2.5 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.

6.3 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 may 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.

6.4 The initial statement

An initial value may also 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* may 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.

7. 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 may break some of the statement forms without an explicit continuation. A square (□) in the syntax represents a point where an end-of-line will be ignored.

7.1 Expression statements

A procedure invocation that returns no value is known as a **subroutine call.** Such an invocation is a statement. Examples are

```
work(in, out)
run( )
```

Input/output statements (see "Input/Output Statements" in this section) 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
```

7.2 Blocks

A **block** is a compound statement that acts as a single statement. A block uses the following syntax:

{ [[*declaration*]] [[*executable-statement*]] }

A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. Figure 12-12 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)
}
```

Figure 12-12. Example of a block

7.3 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

7.3.1 `if` statement

The simplest of the test statements is the `if` statement. Its form is

`if` (*logical-expression*) □ *statement*

where □ means the line may 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.

7.3.2 `if-else`

A more general statement is of the form

`if` (*logical-expression*) □ *statement-1* □
`else` □ *statement-2*

where □ means the line may 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 may 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
```

Figure 12-13. Nested `if-else`

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, shown in Figure 12-14.

```
if(x==1)
    k = 1
else if(x==3  |  x==5)
    k = 2
else
    k = 3
```

Figure 12-14. Sequential `if-else`

You may 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`s, you may prefer to use a select statement instead.

7.3.3 `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
     }
```

Figure 12-15. `select` statement with `case` and `default`

7.4 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.

7.4.1 `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.

7.4.2 `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 Statement" under "Executable Statements"), 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 may be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
        n += i
```

Alternatively, the computation could 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 will be given later.

7.4.3 `repeat` statement

The statement

`repeat` □ *statement*

executes the *statement*, then does it again, without any termination test. A test inside the *statement* is needed to stop the loop.

7.4.4 `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 has not been paired with an `until`. In practice, this appears to be the least frequently used looping construct.

7.4.5 `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* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2*. The sum of the first hundred positive integers could be computed by the following code:

```
n = 0
do i = 1, 100
        n += i
```

7.5 Branch statements

It is not considered good programming practice to use branch statements if you could use a loop construct instead. If you must use a branch statement, however, EFL provides a few for your convenience.

7.5.1 `goto` statement

The most general, and most risky, branching 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 may be used as labels, as in Figure 12-16.

```
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")
        }
```

Figure 12-16. Use of `goto`s with **`case`** labels in a **`select`**

If two `select` statements are nested, the `case` labels of the outer `select` are not accessible from the inner one.

7.5.2 `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/or `select` surrounding the statement.

You may 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
```

will transfer to the statement after the third enclosing `for` loop.

7.5.3 `next` statement

The `next` statement causes the first surrounding loop statement to go on 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:

```
next
next 3
next 3 for
next for 3
```

A `next` statement ignores `select` statements.

7.5.4 `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*)

7.6 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 may 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 will probably be 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 an error. EFL's input/output statements reflect very strongly the facilities of Fortran.

7.6.1 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 may 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.

7.6.2 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* (see below) 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.

7.6.3 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 may include format specifications. If *unit* is omitted, the standard input or output unit is used.

7.6.4 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** may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a `do` statement, and has a similar effect: the values in the braces are transmitted repeatedly until the `do` execution is complete.

7.6.5 Formats

The following 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

Figure 12-17. Permissible format specifiers in EFL

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.

7.6.6 Manipulation statements

The three input/output statements

```
backspace(unit)
rewind(unit)
endfile(unit)
```

look like ordinary procedure calls, but you may 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` will reread the previous record, and the next `write` will overwrite it.

`rewind` moves the device to its beginning, so that the next input statement will read the first record.

`endfile` causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past it will be an error.

8. Procedures

Procedures are the basic unit of an EFL program and provide the means of segmenting a program into separately compilable and named parts.

8.1 `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* may specify precision and type or they may be omitted entirely. You may declare the procedure's precision and type in an ordinary declaration statement. If you do not declare a type, the procedure is a subroutine and no value may 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.

8.2 `end` statement

Each procedure terminates with the statement

```
end
```

8.3 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 may 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 may not change the formal argument's 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.

8.4 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.

8.5 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 upon the attributes of the actual arguments.

8.5.1 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 may 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 result must be `integer`. Sample function calls follow:

```
min(5, x, -3.20)
max(i, z)
```

8.5.2 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.

8.5.3 Elementary functions

Generic functions take arguments of `real`, `long real`, or `complex` type and return a result of the same type:

Table 12-5. Generic functions

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

8.5.4 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.

9. Atavisms

The following constructs are included to ease the conversion of old Fortran programs to EFL.

9.1 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.

9.2 `call` statement

You may precede a subroutine call with the keyword `call`, as follows:

```
call joe
call work(17)
```

9.3 Obsolete keywords

The following keywords are recognized as synonyms of EFL keywords:

Table 12-6. Recognized keyword synonyms

Fortran	EFL
`double precision`	`long real`
`function`	`procedure`
`subroutine`	`procedure` (*untyped*)

9.4 Numeric labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted. The colon is optional following a numeric label.

9.5 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 may 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
```

9.6 Computed `goto`

Fortran contains an indexed multiway branch. You may 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.

9.7 `goto` statement

In unconditional and computed `goto` statements, you may separate the `go` and `to` words, as in

`go to` *xyz*

9.8 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"), that forces the compiler to recognize the forms in the second column in Table 12-6:

Table 12-7. 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 may not name any structure element `lt`, `le`, and so on. The basic forms in the left column, however, are always recognized.

9.9 Complex constants

You may 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)
```

9.10 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 may be assigned, however; an ordinary `return` statement returns the last value assigned to that name as the function value.

9.11 Equivalence

A statement with the form

```
equivalence v1, v2, ..., vn
```

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

9.12 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 may also have variable numbers of arguments, but all the arguments must have the same type. The nongeneric functions are shown in Table 12-7.

Table 12-8. 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

10. Compiler options

You may use a number of options to control the output and tailor it for various compilers and systems. The chosen defaults are conservative, but you may 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.

10.1 Default options

Each option has a default setting. You may 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`.

10.2 Input language options

The `dots` option determines whether the compiler recognizes `.lt.` and similar forms. The default setting is `no`.

10.3 Input/output error handling

The `ioerror` option may be given three values: `none`, `ibm`, or `fortran77`. `none` means that none of the I/O statements may 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.

10.4 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.

10.5 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 may change the default formats by setting certain options:

Table 12-9. Options for changing default **`read/write`** formats

Option	Type
`iformat`	`integer`
`rformat`	`real`
`dformat`	`long real`
`zformat`	`complex`
`zdformat`	`long complex`
`lformat`	`logical`

The associated value must be a Fortran format, such as

```
option rformat=2.6
```

10.6 Alignments and sizes

To implement character variables, structures, and the `sizeof` and `lengthof` operators, you need to know how much space various Fortran data types require and what boundary alignment properties they demand. The relevant options are shown in Table 12-9.

Table 12-10. 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`

The sizes are in terms of an arbitrary unit; the alignment is in the same unit. The option `charperint` gives the number of characters per `integer` variable.

10.7 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`.

10.8 Miscellaneous output control options

Each Fortran procedure the compiler generates will be preceded by the value of the `procheader` option.

No Hollerith strings will be passed as subroutine arguments if `hollincall=no` is specified.

The Fortran statement numbers normally start at one and increase by one. You may change the increment value by using the `deltastno` option.

11. Examples

The following short examples of EFL programming show some of the convenience of the language.

11.1 File copying

This short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure    # main program
character(100) line

while(read(, line) == 0)
    write(, line)
end
```

Figure 12-18. File-copying example

Because `read` returns zero until the end-of-file (or a `read` error), this program keeps reading and writing until the input is exhausted.

11.2 Matrix multiplication

This procedure 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

$$c_{ij} = \Sigma\, a_{ik}\, b_{kj}$$

```
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
```

Figure 12-19. Matrix multiplication example

11.3 Searching a linked list

If you have a list of number pairs (x, y), that list is stored as a linked list, sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value:

```
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
```

Figure 12-20. Example of searching a linked list

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`.

11.4 Walking a tree

An example of a more complicated problem would be if you had an expression tree stored in a common area and you wanted 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 would be implemented by the following simple pseudocode:

```
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
```

Figure 12-21. Pseudocode for a tree walk

In a nonrecursive language like EFL, you need to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure `outch` to print a single character and a procedure `outval` to print a value:

```
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
```

Figure 12-22. Example of walking a tree (page 1 of 2)

```
#  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
                }
            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
```

Figure 12-22. Example of walking a tree (page 2 of 2)

12. 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).

12.1 Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

12.1.1 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`.

12.1.2 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.

13. Compiler

13.1 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 above except for `long complex` numbers.

13.2 Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) in which the error was detected. Warnings are given for variables that are used but not explicitly declared.

13.3 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). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (see "Examples"):

```
      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 12-23. Fortran code produced from matrix multiplication example

The following is the procedure for the tree-walk:

```
      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()
```

Figure 12-24. Fortran code produced from tree-walk example (page 1 of 2)

```
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
```

Figure 12-24. Fortran code produced from tree-walk example (page 2 of 2)

14. Constraints on EFL

Although Fortran may 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.

14.1 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.

14.2 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 may only be passed as an argument or invoked; it cannot be stored.

14.3 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 could have been handled by pointers. Although there are several ways of simulating pointers by using subscripts, this raises problems of external variables and initialization.

14.4 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 may be simulated in EFL, but not without considerable effort.

14.5 Storage allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using common blocks.

Chapter 13
`as` Reference

Contents

Tables

Chapter 13

as Reference

1. as: The assembler

Programmers familiar with the M68000 family of processors should be able to program in the A/UX resident assembler, as, by referring to this chapter, but this is not a reference for the processor itself. Details about the effects of instructions, meanings of status register bits, handling of interrupts, and many other issues are not dealt with here. This chapter, therefore, should be used in conjunction with the following reference manuals:

- *M68000 16/32-Bit Microprocessor Programmer's Reference Manual,* Fourth Edition; Englewood Cliffs, N. J.: Prentice-Hall, 1984. This manual is also available from the Motorola Literature Distribution Center, part number M68000UM.
- *MC68020 32-Bit Microprocessor User's Manual;* Englewood Cliffs, N. J.: Prentice-Hall, 1984. This manual is also available from the Motorola Literature Distribution Center, part number MC68020UM.
- *MC68851 Paged Memory Management Unit User's Manual,* part number MC68851UM/AD.
- *MC68881 Floating Point Coprocessor User's Manual,* part number MC68881UM/AD.
- *M68000 Family Resident Structured Assembler Reference Manual,* part number M68KMASM.
- *A/UX User Interface.*
- *A/UX Command Reference.*

2. Warnings

A few important warnings to the `as` user should be emphasized at the outset. Although, for the most part, there is a direct correspondence between `as` notation and the notation used in the documents listed in the preceding section, several exceptions exist that could lead the unsuspecting user to write incorrect code. In addition to the exceptions described in the following paragraphs, refer also to the sections "Address mode syntax" and "Machine instructions" for further information.

2.1 Comparison instructions

First, the order of the operands in `compare` instructions follows one convention in the *M68000 Programmer's Reference Manual* and the opposite convention in `as`. Using the convention of the *M68000 Programmer's Reference Manual*, one might write

```
CMP.W   D5, D3    Is D3 less than D5?
BLE     IS_LESS   Branch if less.
```

Using the `as` convention, one would write

```
cmp.w   %d3,%d5   # Is d3 less than d5 ?
ble     is_less   # Branch if less.
```

This convention 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 may 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 the convention find that both the compare and subtract notations make sense in their respective contexts.

2.2 Case

In the A/UX implementation, only lowercase instruction and register names are valid. For example,

```
mov    %d1,%d2   # works
```

is in an acceptable case, while

```
MOV   %D1,%D2  # does not work
```

is not. This is especially important for those who wish to port existing code from other machines.

2.3 Overloading of opcodes

Another issue that users must be aware of arises from the M68000 processors' use of several different instructions to do more or less the same thing. For example, the *M68000 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 M68000 operation code. The danger created by this convenience is that it could give the misleading impression that all forms of the `SUB` operation are semantically identical. In fact, they are not. The careful reader of the *M68000 Programmer's Reference Manual* will notice that whereas `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 will not be affected.

3. Using `as`

The A/UX command `as` invokes the assembler and has the following syntax:

`as [-m][-n][-o` *outfile*`][-R][-V]` *filename*

The following flags may be specified in any order:

`-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 a `.o` suffix.

`-n` — Turn off long/short address optimization. By default, address optimization takes place.

`-m` Run the `m4` macro pre-processor on the input to the assembler.

> *Note:* If the `-m` flag option is used, keywords for `m4` cannot be used as symbols (variables, functions, labels) in the input file because `m4` cannot determine which are assembler symbols and which are real `m4` macros.

`-R` Remove (unlink) the input file after assembly is completed. This flag option is off by default.

`-V` Write the version number of the assembler being run on the standard error output.

4. General syntax rules

4.1 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**. One or more labels may precede any assembly language instruction or pseudooperation. Refer to "Location Counters and Labels" below.

- A line of assembly code need not include an instruction. It may consist of a comment alone (introduced by #), or a label alone (terminated by :), or it may 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 may appear at the beginning of the line, if desired, and spaces may 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.

4.2 Comments

Comments are introduced by the character # and continue to the end of the line. Comments may appear anywhere and are disregarded by the assembler.

4.3 Identifiers

An identifier is 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. Uppercase and lowercase letters are distinguished; for example, `con35` and `CON35` are two distinct identifiers.

There is no limit on the length of an identifier, except as imposed by the loader on the system.

The value of an identifier is established by the `set` pseudooperation (refer to "Symbol Definition Operations") or by using it as a label (refer to "Location Counters and Labels").

The tilde character (~) has special significance to the assembler. A ~ used alone, as an identifier, means "the current location." A ~ used as the first character in an identifier becomes a period (`.`) in the symbol table, allowing symbols such as `.eos` and `.0fake` to be entered into the symbol table, as required by the Common Object File Format

(COFF). Information about file formats is provided in Section 4 of *A/UX Programmer's Reference*.

4.4 Register Identifiers

A register identifier is an identifier preceded by the character `%`, and represents one of the MC68000 processor's registers. The predefined register identifiers are

```
%d0   %d4   %a0   %a4   %cc   %usp
%d1   %d5   %a1   %a5   %pc   %fp
%d2   %d6   %a2   %a6   %sp
%d3   %d7   %a3   %a7   %sr
```

Note: The identifiers `%a7` and `%sp` 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 may result in confusion.

The current version of the assembler will correctly assemble instructions intended for the M68010. The following additions will be flagged with warnings:

Registers added for the MC68010	
Name	**Description**
`%sfc,%sfcr`	Source function code register
`%dfc,%dfcr`	Destination function code register
`%vbr`	Vector base register

- `%sfc` and `%sfcr` are equivalent.
- `%dfc` and `%dfcr` are equivalent.

The entire register set of the MC68000 and MC68010 is included in the MC68020 register set. The following are new control registers for the MC68020:

MC68020 registers	
Name	**Description**
`%caar`	Cache address register
`%cacr`	Cache control register
`%isp`	Interrupt stack pointer
`%msp`	Master stack pointer

The following are suppressed registers (zero registers) used in various MC68020 addressing modes:

MC68020 zero registers		
Suppressed address registers	**Suppressed data registers**	**Suppressed program counter**
`%za0`	`%zd0`	`%zpc`
`%za1`	`%zd1`	
`%za2`	`%zd2`	
`%za3`	`%zd3`	
`%za4`	`%zd4`	
`%za5`	`%zd5`	
`%za6`	`%zd6`	
`%za7`	`%zd7`	

4.5 Constants

`as` deals only with integer constants. They may be entered in decimal, octal, or hexadecimal, or they may be entered as character constants. Internally, `as` treats all constants as 32-bit binary 2's-complement quantities.

4.5.1 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.

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)
```

4.5.2 Character constants

An ordinary character constant consists of a single-quote character (') followed by an arbitrary ASCII character other than the backslash (\). The value of the constant is equal to the ASCII code for the character. Special meanings of characters are overridden when used in character constants; for example, if '# is used, the # is not treated as introducing a comment.

A special character constant consists of '\ followed by another character. All the special character constants and examples of ordinary character constants are listed in the following table.

Table 13-1. Ordinary and 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
''	0x27	Single quote
'0	0x30	Zero
'A	0x41	Uppercase A
'a	0x61	Lowercase a

4.6 Other syntactic details

A discussion of expression syntax appears in "Expressions". Information about the syntax of specific components of `as` instructions and pseudooperations is given in "Pseudooperations," "Span-dependent Optimization," and "Address Mode Syntax," below.

5. Segments, location counters, and labels

5.1 Segments

A program in `as` assembly language may 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.

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 may switch freely among them by using assembler pseudooperations (refer to "Location Counter Control Operations," below). 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 may not be the same as in the assembly language file. For example, if the data for a program consisted of

```
data    1               # segment 1
short   0x1111
data    0               # segment 0
long    0xffffffff
data    1               # segment 1
byte    0xff
```

then equivalent object code would be generated by

```
data    1
data    0
long    0xffffffff
short   0x1111
byte    0xff
```

5.2 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 one 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.

6. Types

Identifiers and expressions may have values of different types.

- In the simplest case, an expression or identifier may have an **absolute value,** such as 29, -5000, or 262143.
- An expression or identifier may 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 may be used with the expectation that their values will be defined in another program, and therefore known at load time; but the relative values of undefined externals cannot be known.

7. Expressions

For conciseness, the following abbreviations are useful:

abs	absolute expression
rel	relocatable expression
ext	undefined external

All constants are absolute expressions. An identifier may be thought of as an expression having the identifier's type. Expressions may 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 (refer to "Switch Table Operation" below). Use of a *rel* – *rel* expression is dangerous, particularly when dealing with identifiers from `text` segments. The problem is that the assembler will determine the value of the expression before it has resolved 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 may be used to coerce the order of evaluation.

If the result of a division is a positive noninteger, it will be truncated toward zero. If the result is a negative noninteger, the direction of truncation cannot be guaranteed.

8. Pseudooperations

8.1 Data initialization operations

`byte` *abs, abs, ...*

One or more arguments, separated by commas, may 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, may 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 arguments, separated by commas, may be given. Each expression may 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 may not be filled in until load time). Alternatively, the arguments may be bit-field expressions. A bit-field 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 bit field. Successive bit fields fill up 32-bit `long` quantities, starting with the high-order part. If the sum of the lengths of the bit fields is less than 32 bits, the assembler creates a 32-bit `long` with zeros 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
```

Bit fields may not span pairs of 32-bit `long`s. Thus,

```
long    24: 0xa, 24: 0xb, 24:0xc
```

yields the same thing as

```
long    0x00000a00, 0x00000b00, 0x00000c00
```

`space` *abs*

The value of *abs* is computed, and the resultant number of bytes of zero data is generated. For example,

```
space   6
```

is equivalent to

```
byte    0,0,0,0,0,0
```

8.2 Symbol definition operations

`set` *identifier, expr*

The value of *identifier* is set equal to *expr*, which may be absolute or relocatable.

`comm` *identifier, abs*

The named identifier is to be assigned to a common area of size *abs* bytes. If *identifier* is not defined by another program, the loader will allocate space for it.

`lcomm` *identifier, abs*

The named *identifier* is assigned to a local common area of size *abs* bytes. This results in allocation of 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, then 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; in this case, therefore, *identifier* is global by default.

8.3 Location counter control operations

`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*. *expr* 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.

8.4 Symbolic debugging operations

The assembler allows for symbolic debugging information to be placed into the object code file with special pseudooperations. The information typically includes line numbers and information about C language symbols, such as their type and storage class. The C compiler (`cc`(1)) generates symbolic debugging information when the `-g` flag option is used. Assembler programmers may also include such information in source files.

8.4.1 `file` and `ln`

The `file` pseudooperation passes the name of the source file into the object file symbol table. It has the form

`file` *filename*

where *filename* consists of 1 to 14 characters enclosed in quotation marks.

The `ln` pseudooperation 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 `text`, `data`, or `bss` to associate with the line number. The default when *value* is omitted (which is usually the case) is the current location in `text`.

8.4.2 Symbol attribute operations

The basic symbolic testing pseudooperations are `def` and `endef`. These operations enclose other pseudooperations 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* may be any one of the attribute assigning operations shown below.

`def` does not define the symbol, although it does create a symbol table entry. Because an undefined symbol is treated as external, a symbol which appears in a `def` but which never acquires a value will ultimately result in an error at link edit 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 paragraphs below describe the attribute-assigning operations (*attrasgn* in the above syntax diagram). Keep in mind that all these operations apply to the symbol *name* that appeared in the opening `def` pseudooperation.

`val` *expr* — 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 — Declares a storage class for *name*. The expression *expr* must yield an absolute value that corresponds to the C compiler's internal representation of a storage class. The special value –1 designates the physical end of a function.

`type` *expr* — Declares the C language type of *name*. The expression *expr* must yield an absolute value that corresponds to the C compiler's internal representation of a basic or derived type.

`tag` *str* — Associates *name* with the structure, enumeration, or union named *str* that must have already been declared with a `def/endef` pair.

`line` *expr* — 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* — 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 bit fields, the size is in bits.

`dim` *expr1*`,` *expr2*`, ...` — Indicates that *name* is an array. Each of the expressions must yield an absolute value that provides the corresponding array dimension.

8.5 Switch table operation

The C compiler generates a compact set of instructions for the C language `switch` construct. An example is shown below.

```
        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` pseudooperation communicates to the assembler that the lines following it contain *rel* – *rel* subtractions. Remember that 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 will be defined in the current assembler file, and that one of the symbols is a fixed distance away from the current location.

The `swbeg` pseudooperation takes an argument that looks like an immediate operand. The argument is the number of lines that follow `swbeg` and 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, it gets confused when it tries to decode the table entries, differences between two symbols, as instructions.

9. Span-dependent optimization

The assembler makes certain choices about the object code it generates based on the distance between an instruction and its operand(s). 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 may 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` flag 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 compiler (`cc`(1)) generates branch instructions without a specific offset size. When the optimizer is used, it identifies branches that could 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 could have need for a branch to a location not reachable 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, then the word form of the instruction will be generated. Otherwise, the very long form will be 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 address operand (instead of `pc`-relative). For conditional branches, the equivalent very long form is a conditional branch around a jump, where the conditional test has been reversed.

The following table summarizes span-dependent optimizations. 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 13-2. 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

For the MC68020 processor, branch instructions may 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.

10. Address mode syntax

The following tables summarize the `as` syntax for MC68020 addressing modes:

In the tables, the following abbreviations are used:

`An/an` Address register, where `n` is any digit from 0 through 7.

bd 2's-complement base displacement that is added before indirection takes place; size may be 16 or 32 bits.

d 2's-complement or sign-extended displacement that is added as part of effective address calculation; size may be 8 or 16 bits; when omitted, assembler uses value of zero.

`Dn/dn` 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; size may be 16 or 32 bits.

`PC/pc` Program counter.

Ri/ri Index register `i` may be any address or data register with an optional size designation (that is, *ri*`.w` for 16 bits or *ri*`.l` for 32 bits); default size is `.w`.

scl Optional scale factor that may be multiplied times index register in some modes. Values for *scl* are 1, 2, 4, or 8; default is 1.

`[ ]` Grouping characters used to enclose an indirect expression; required characters. Addressing arguments may occur in any order within the brackets.

`( )` Grouping characters used to enclose an entire effective address; required characters. Addressing arguments may occur in any order within the parentheses.

`{ }` Indicate that a scale factor is optional; not required characters.

It is important to note that expressions used for the absolute addressing modes need not be absolute expressions in the sense described in "Types," above. Although the addresses used in those addressing modes ultimately must be filled in with constants, that can be done later by the loader. There is no need for the assembler to be able to compute them. Indeed, the absolute long addressing mode is commonly used for accessing undefined external addresses.

Table 13-3. Effective address modes

M680x0 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)

Table 13-4. Effective address modes (continued)

M680x0 notation	as notation	Address mode
d(PC,*Ri*)	*d*(%pc,%*rn.l*) *d*(%pc,%*rn*.w)	Program pounter 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 post-indexing plus base and outer displacement
([*bd*,PC,*Ri*{**scl*}],*od*)	([*bd*,%pc,%*ri*{**scl*}],*od*)	Program counter memory indirect with prein-dexing 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)
#*xxx*	&*xxx*	Immediate data (*xxx* signifies an absolute constant expression)

In the table above, the index register notation should be understood as *ri*.*size***scale*, where both *size* and *scale* are optional. Refer to Chapter 2 of the *M68000 Family Resident Structured Assembler Reference Manual* for additional information about effective address modes. Section 2 of the *MC68020 32-Bit Microprocessor User's Manual* also provides information about generating effective addresses and assembler syntax.

Note that suppressed address register `%za`*n* may be used in place of `%a`*n*, suppressed PC register `%zpc` may be used in place of `%pc`, and suppressed data register `%zd`*n* may be used in place of `%d`*n*, if suppression is desired.

The new 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 will generate the brief format if the effective address expression is not memory indirect, value of displacement is within a byte, and no base or index suppression is specified; otherwise, the assembler will generate the full format.

Some source code variations of the new modes may be redundant with the MC68000 address register indirect, address register indirect with displacement, and program counter with displacement modes. The assembler will select the more efficient mode when redundancy occurs. For example, when the assembler sees the form `(A`*n*`)`, it will generate address register indirect mode (mode 2).

The assembler will generate address register indirect with displacement (mode 5) when seeing any of the following forms (as long as *bd* fits in 16 bits or less):

bd`(A`*n*`)`
`(`*bd*`,A`*n*`)`
`(A`*n*`,`*bd*`)`

11. Machine instructions

The following table shows how MC68020 instructions should be written in order to be understood correctly by the `as` assembler.

Several abbreviations are used in the table:

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` may not be used in the `b`*CC* instruction):

Table 13-5. Condition code designations

`cc`	Carry clear	`ls`	Low or same
`cs`	Carry set	`lt`	Less than
`eq`	Equal	`mi`	Minus
`f`	False	`ne`	Not equal
`ge`	Greater or equal	`pl`	Plus
`gt`	Greater than	`t`	True
`hi`	High	`vc`	Overflow clear
`hs`	High or same (=`cc`)	`vs`	Overflow set
`le`	Less or equal		
`lo`	Low (=`cs`)		

d — 2's-complement or sign-extended displacement that is added as part of effective address calculation; size may be 8 or 16 bits; when omitted, assembler uses value of zero.

EA — An arbitrary effective address.

(eq) — The two forms of machine instruction are equivalent.

I — An absolute expression, used as an immediate operand.

L — A label reference, or any expression representing a memory address in the current segment.

offset — Either an immediate operand or a data register.

Q — An absolute expression evaluating to a number from one to eight.

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.
`%`*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.

MC68000 instruction formats

Mnemonic	Assembler syntax	Operation
ABCD	`abcd.b %dy,%dx` `abcd.b -(%ay),-(%ax)`	Add decimal with extend
ADD	`add.`*S* *EA*`,%d`*n* `add.`*S* `%d`*n*`,`*EA*	Add binary
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* `%dy,%dx` `addx.`*S* `-(%ay),-(%ax)`	Add extended
AND	`and.`*S* *EA*`,%d`*n* `and.`*S* `%d`*n*`,`*EA*	AND logical
ANDI	`and.`*S* `&`*I*`,`*EA*	AND immediate
ANDI to CCR	`and.b &`*I*`,%cc`	AND immediate to condition codes
ANDI to SR	`and.w &`*I*`,%sr`	AND immediate to the status register

ASL	`asl.`*S* `%dx,%dy`
	`asl.`*S* `&`*Q*`,%dy`
	
	`asl.w &1,`*EA*
		Arithmetic shift (left)

ASR	`asr.`*S* `%dx,%dy`
	`asr.`*S* `&`*Q*`,%dy`
	
	`asr.w &1,`*EA*
		Arithmetic shift (right)

MC68000 instruction formats

Mnemonic	Assembler syntax	Operation
Bcc	b*CC* *L*	Branch conditionally (16-bit displacement)
	b*CC*.b *L*	Branch conditionally (short) (8-bit displacement)
	b*CC*.l *L*	Branch conditionally (long) (32-bit displacement)
BCHG	bchg %d*n*,*EA* bchg &*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.
BCLR	bclr %d*n*,*EA* bclr &*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.
BFCHG	bfchg *EA*{*offset:width*}	Complement bit field
BFCLR	bfclr *EA*{*offset:width*}	Clear bit field
BFEXTS	bfexts *EA*{*offset:width*},%d*n*	Extract bit field (signed)
BFEXTU	bfextu *EA*{*offset:width*},%d*n*	Extract bit field (unsigned)
BFFFO	bfffo *EA*{*offset:width*},%d*n*	Find first one in bit field
BFINS	bfins %d*n*, *EA*{*offset:width*}	Insert bit field

MC68000 instruction formats

Mnemonic	Assembler syntax	Operation
BFSET	`bfset` *EA*{*offset*:*width*}	Set bit field
BFTST	`bftst` *EA*{*offset*:*width*}	Test bit field
BKPT	`bkpt` `&`*I*	Breakpoint
BRA	`bra.`*S* *L*	Branch always
	`br.`*S* *L*	Same as `bra.`*S*
BSET	`bset` `%d`*n*,*EA* `bset` `&`*I*,*EA*	Test a bit and set *Note:* `bset` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is.
BSR	`bsr.`*S* *L*	Branch to subroutine
BTST	`btst` `%d`*n*,*EA* `btst` `&`*I*,*EA*	Test a bit and set *Note:* `btst` must be written with no suffix. If the second operand is a data register, `.l` is assumed; otherwise, `.b` is.
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*, (`%r`*x*):(`%r`*y*)	Compare and swap dual operands

MC68000 instruction formats

Mnemonic	Assembler syntax		Operation
CHK	`chk.`*A*	*EA*, `%d`*n*	Check register against bounds
CHK2	`chk2.`*S*	*EA*, `%r`*n*	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*	`%r`*n*, *EA*	Compare register against bounds†
DBcc	`db`*CC*	`%d`*n*, *L*	Test condition, decrement, and branch
	`dbra`	`%d`*n*, *L*	Decrement and branch always
	`dbr`	`%d`*n*, *L*	Same as `dbra`
DIVS	`divs.w`	*EA*, `%d`*x*	Signed divide 32/16–> 16r:16q
	`tdivs.l`	*EA*, `%d`*x*	Signed divide (long)
	`divs.l`	*EA*, `%d`*x*	32/32 –> 32q
	`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

* The order of operands in `as` is the reverse of that in the *M68000 Programmer's Reference Manual.*

† The `cmp.`*S* syntax is also recognized.

‡ Whenever `%d`*x* and `%d`*y* are the same register, then the form is equivalent to the `tdivs.l` *EA*, `%d`*x* form.

MC68000 instruction formats

Mnemonic	Assembler syntax	Operation
DIVU	`divu.w` *EA*, `%d`*n*	Unsigned divide 32/16 –> 16r:16q
DIVUL	`tdivu.l` *EA*, `%d`*x* `divu.l` *EA*, `%d`*x*	Unsigned divide (long) 32/32 –> 32*(eq)*
	`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†
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` `%r`*x*, `%r`*y*	Exchange registers
EXT	`ext.w` `%d`*n*	Sign-extend low-order Byte of data to word
EXTB	`ext.l` `%d`*n*	Sign-extend low-order Word of data to long
	`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

* Whenever `%d`*x* and `%d`*y* are the same register, then the form is equivalent to the `divu.l` *EA*, `%d`*x* form.

† Whenever `%d`*x* and `%d`*y* are the same register, then the form is equivalent to the `tdivu.l` *EA*, `%d`*x* form.

MC68000 instruction formats

Mnemonic	Assembler syntax		Operation
LEA	`lea`	*EA*, `%a`*n*	Load effective Address
LINK	`link.`*A*	`%a`*n*, `&`*I*	Link and allocate

LSL	`lsl.`*S*
	`lsl.`*S*
	`lsl.`*S*
		`%d`*x*, `%dy`
		`&`*Q*, `%dy`
		EA
			Logical shift (left)

LSR	`lsr.`*S*
	`lsr.`*S*
	`lsr.`*S*
		`%d`*x*, `%dy`
		`&`*Q*, `&dy`
		EA
			Logical shift (right)

MOVE	`move.`*S*	*EA*,*EA*	Move data from source to destination*†
MOVE to CCR	`move.w`	*EA*, `%cc`	Move to condition codes*
MOVE from CCR	`move.w`	`%cc`,*EA*	Move from condition codes*
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` `move.l`	`%usp`, `%a`*n* `%a`*n*, `%usp`	Move user stack pointer*
MOVEA	`move.`*A*	*EA*, `%a`*n*	Move address*
MOVEC	`move.l` `move.l`	`%`*rc*, `%`*rn* `%`*rn*, `%`*rc*	Move from/to control register*

* In all move commands, `move` may be shortened to `mov`.

† If the destination is an address register, the instruction generated is `MOVEA`.

MC68000 instruction formats

Mnemonic	Assembler syntax	Operation
MOVEM	`movem.`*A* *EA*`,&`*I* `movem.`*A* `&`*I*`,`*EA*	Move multiple registers*†
MOVEP	`movep.`*A* `%dx,`*d*`(%ay)` `movep.`*A* *d*`(%ay),%dx`	Move peripheral data*
MOVEQ	`move.l` `&`*I*`,%d`*n*	Move quick*
MOVES	`moves.`*S* `%`*rn*`,`*EA* `moves.`*S* *EA*`,%`*rn*	Move to/from address space*
MULS	`muls.w` *EA*`,%dx`	Signed multiply 16*16 –> 32
	`tmuls.l` *EA*`,%dx`	Signed multiply (long)
	`muls.l` *EA*`,%dx`	32*32 –> 32 *(eq)*
	`muls.l` *EA*`,%dx:%dy`	Signed multiply (long) 32*32 –> 64
MULU	`mulu.w` *EA*`,%dx`	Unsigned multiply 16*16 –> 32
	`tmulu.l` *EA*`,%dx`	Unsigned multiply (long)
	`mulu.l` *EA*`,%dx`	32*32 –> 32*(eq)*
	`mulu.l` *EA*`,%dx:%dy`	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

* In all 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 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.

MC68000 instruction formats

Mnemonic	Assembler syntax	Operation
OR	`or.`*S* *EA*,`%d`*n* `or.`*S* `%d`*n*,*EA*	Inclusive OR logical
ORI	`ori.`*S* `&`*I*,*EA*	Inclusive OR immediate. Equivalent to `or.`*S*
ORI to CCR	`ori.w` `&`*I*,`%cc`	Inclusive OR immediate to Condition codes. 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` `%d`*x*`,%d`*y*`,&`*I*	Pack BCD
PEA	`pea` *EA*	Push effective address
RESET	`reset`	Reset external devices
ROL	`rol.`*S* `%d`*x*`,%d`*y* `rol.`*S* `&`*Q*`,%d`*y* `rol.w` *EA*	Rotate (without extend) (Left)
ROR	`ror.`*S* `%d`*x*`,%d`*y* `ror.`*S* `&`*Q*`,%d`*y* `ror.w` *EA*	Rotate (without extend) (right)
ROXL	`roxl.`*S* `%d`*x*`,%d`*y* `roxl.`*S* `&`*Q*`,%d`*y* `roxl.w` *EA*	Rotate with extend (left)
ROXR	`roxr.`*S* `%d`*x*`,%d`*y* `roxr.`*S* `&`*Q*`,%d`*y* `roxr.w` *EA*	Rotate with extend (right)
RTD	`rtd` `&`*I*	Return and deallocate parameters
RTE	`rte`	Return from exception
RTM	`rtm` `%r`*n*	Return from module
RTR	`rtr`	Return and restore condition codes
RTS	`rts`	Return from subroutine

MC68000 instruction formats

Mnemonic	Assembler syntax		Operation
SBCD	`sbcd`	`%dy,%dx`	Subtract decimal with extend
	`sbcd`	`-(%ay),-(%ax)`	
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*	`%dy,%dx`	Subtract with extend
		`-(%ay),-(%ax)`	
SWAP	`swap`	`%d`*n*	Swap register halves
TAS	`tas`	*EA*	Test and set an operand
TRAP	`trap`	*&I*	Trap
TRAPV	`trapv`		Trap on overflow
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`	`-(%ax),-(%ay),&`*I*	Unpack BCD
	`unpk`	`%dx,%dy,&`*I*	

11.1 Instructions for the MC68881

The following table shows how the floating-point coprocessor (MC68881) instructions should be written to be understood by the `as` assembler.

In the table, *CC* represents any of the following floating-point condition code designations.

Table 13-6. TRAP on unordered

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 13-7. No TRAP on unordered

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 designation *ccc* represents a group of constants in MC68881 constant ROM that have the following values:

Table 13-8. Constants in MC68881 constant 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

Additional abbreviations used in the table are

A	Source format letters `w` or `l`
B	Source format letters `b`, `w`, `l`, `s`, or `p`
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
`%`*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

Note: The source format must be specified if more than one source format is permitted, or a default source format `x` is assumed. Source format need not be specified if only one format is permitted by the operation.

MC68881 instruction formats

Mnemonic	Assembler syntax		Operation
FABS	`fabs.`*SF*	*EA*, `%fp`*n*	Absolute value function
	`fabs.x`	`%fp`*m*, `%fp`*n*	
	`fabs.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.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*	

* The order of operands in `as` is the reverse of that in the M68000 Programmer's Reference Manual.

MC68881 instruction formats

Mnemonic	Assembler syntax		Operation
FDBcc	`fdb`*CC*`.w`	`%d`*n*`,`*L*	Decrement and branch on condition
FDIV	`fdiv.`*SF*	*EA*`,%fp`*n*	Floating-point divide
	`fdiv.x`	`%fp`*m*`,%fp`*n*	
FETOX	`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*	

MC68881 instruction formats

Mnemonic	Assembler syntax		Operation
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	`fmove.`*SF*	*EA*, `%fp`*n*	Move to floating-point register*
	`fmove.x`	`%fp`*m*, `%fp`*n*	
	`fmove.`*SF*	`%fp`*n*, *EA*	Move from floating-point register to memory*
	`fmove.p`	`%fp`*n*, *EA*`{&`*I*`}`	
	`fmove.p`	`%fp`*n*, *EA*`{%d`*n*`}`	
FMOVE (cont'd.)	`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*	

* In all (floating-point) move commands, `move` may be shortened to `mov`.

MC68881 instruction formats

Mnemonic	Assembler syntax		Operation
FMOVECR	`fmovcr.x`	*`&ccc`*`,%fp`*`n`*	Move a ROM-stored 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.x`	`%fp`*`m`*`,%fp`*`n`*	
FNEG	`fneg.`*`SF`*	*EA*`,%fp`*`n`*	Negate function
	`fneg.x`	`%fp`*`m`*`,%fp`*`n`*	
	`fneg.x`	`%fp`*`n`*	

* 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 13-7, Constants in MC68881 constant ROM, in "Instructions for the MC68881."

MC68881 instruction formats

Mnemonic	Assembler syntax		Operation
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
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*	*EA*`,%fp`*n*	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
	`fsqrt.x`	`%fp`*m*`,%fp`*n*	
	`fsqrt.x`	`%fp`*n*	
FSUB	`fsub.`*SF*	*EA*`,%fp`*n*	Square root function
	`fsub.x`	`%fp`*m*`,%fp`*n*	

MC68881 instruction formats

Mnemonic	Assembler syntax		Operation
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
FTRAPcc	`ftrap`*CC*		Trap on condition without a parameter
FTPcc	`ftp`*CC*`.`*A*	&*I*	Trap on condition with a parameter
FTRAPcc	`ftrap`*CC*`.`*A*	&*I*	Trap on condition with a parameter
FTST	`ftest.`*SF*	*EA*	Floating-point test an operand *Note:* 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).
	`ftest.x`	`%fp`*m*	
	`ftst.`*SF*	*EA*	
	`ftst.x`	`%fp`*m*	
FTWOTOX	`ftwotox.`*SF*	*EA*, `%fp`*n*	2**x function
	`ftwotox.x`	`%fp`*m*, `%fp`*n*	
	`ftwotox.x`	`%fp`*n*	

11.2 Instructions for the MC68851

The following table shows how the paged memory management unit (PMMU) (MC68851) instructions should be written to be understood by the `as` assembler.

In the table, *CC* represents any of the following condition code designations:

SET PSR BIT	
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

CLEAR PSR BIT	
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 the table are

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* Represents an absolute expression used as an immediate operand
- `%dfc` Represents the destination function code register
- `%d`*n* Represents a data register
- `%sfc` Represents the source function code register
- `%sfcr` Represents 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` Represents PMMU access control register
- `%bac` Represents PMMU breakpoint acknowledge control register 0 through 7
- `%bad` Represents PMMU breakpoint acknowledge data register 0 through 7
- `%cal` Represents PMMU current access level register
- `%crp` Represents PMMU CPU root pointer register
- `%drp` Represents PMMU DMA root pointer register
- `%pcsr` Represents PMMU cache status register

`%psr`	Represents PMMU status register
`%scc`	Represents PMMU stack change control register
`%srp`	Represents PMMU supervisor root pointer register
`%tc`	Represents PMMU transition control register
`%val`	Represents PMMU validate access level register

Note: The source format must be specified if more than one source format is permitted or a default source format of `w` is assumed. Source format need not be specified if only one format is permitted by the operation.

MC68851 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 pflush	*FC*, &*M* *FC*, &*M*, *EA*	Invalidate entries in ATC
PFLUSHA	pflusha		Invalidate all ATC entries
PFLUSHS	pflushs pflushs	*FC*, &*M* *FC*, &*M*, *EA*	Invalidate entries in ATC including shared entries
PFLUSHR	pflushr	*EA*	Invalidate ATC and RPT entries
PLOADR	ploadr	*FC*, *EA*	Load an entry into ATC
PLOADW	ploadw	*FC*, *EA*	Load an entry into ATC
PMOVE	pmove.*A* pmove.*A*	%*pm*, *EA* *EA*, %*pm*	Move PMMU register*
PRESTORE	prestore	*EA*	PMMU restore function
PSAVE	psave	*EA*	PMMU save function
PS*cc*	ps*CC*	*EA*	Set on PMMU condition

* The pmov. syntax is also recognized.

MC68851 instruction formats

Mnemonic	Assembler syntax		Operation
PTESTR	`ptestr`	*FC,EA,&D*	Get information about logical address
	`ptestr`	*FC,EA,&D,%an*	
PTESTW	`ptestw`	*FC,EA,&D*	Get information about logical address
	`ptestw`	*FC,EA,&D,%an*	
PTRAPcc	`pt`*CC*		Trap on PMMU condition
	`ptrap`*CC*		
	`pt`*CC.A*	*&I*	
	`ptrap`*CC.A*	*&I*	
PVALID	`pvalid`	`%val`,*EA*	Validate a pointer
	`pvalid`	`%a`*n*,*EA*	

Chapter 14
`ld` Reference

Contents

Tables

Chapter 14

ld Reference

1. ld: The link editor

The link editor ld 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 (see ld(1)).

ld supports a command language that lets you control the linking process with great flexibility and precision. Although the link edit process is controlled in detail through use of this language (described later), most users do not require this degree of flexibility, and the manual page ld(1) in *A/UX Command Reference* is sufficient instruction in the use of this command.

The command language allows the link editor

- to specify the machine's memory configuration
- to combine object file sections in particular fashions
- to cause the files to be bound to specific addresses or within specific portions of memory
- to define or redefine global symbols at link edit time

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 file's **magic number** (the first two bytes of the file) to determine which type of file it is encountering. 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 linked together to form an output object file. If there are no unresolved references, you may execute this file on the target machine.

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 link the object files *file1*`.o` and *file2*`.o`, this command is enough:

`ld` *file1*`.o` *file2*`.o`

No directives to `ld` are needed. If no errors are encountered during the link edit, the output is left in the default file `a.out`.

The input file sections are combined 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. The output `.text` section is a concatenation of `.text` from *file1*`.o` and *file2*`.o`. The `.data` and `.bss` sections are formed similarly. The output `.text` section is then bound at address 0x000000. The output `.data` and `.bss` sections are link edited together into contiguous addresses, the particular address depending on the particular processor.

An input file containing link editor directives is referred to as an **i-file** in this document. Its usefulness is explained below. An i-file named `default.ld` is searched for automatically in the list of library directories (see the `-l` and `-L` options under "Options"). The default directory for this search is `/usr/lib`.

Instead of entering the names of files to be link edited, or entering `ld` options on the `ld` command line, you may place this information in an i-file and just pass the i-file to `ld`. For example, if you are frequently going to link the object files *file1*`.o`, *file2*`.o`, and *file3*`.o` with the same options *f1* and *f2*, you might enter the command

`ld` *-f1 -f2 file1*`.o` *file2*`.o` *file3*`.o`

each time you have to invoke `ld`. Alternatively, you could create an i-file containing the statements

-f1
-f2
file1.o
file2.o
file3.o

and use the following command:

`ld` *i-file*

Note that it is perfectly permissible to specify some of the object files to be link edited in the i-file and to specify others on the command line, as well as specifying some options in the i-file and others on the command line. Input object files are link edited 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 link editing 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.

1.1 Some general points

There are several concepts and definitions with which you should become familiar before you proceed further.

1.1.1 Host and target machine

In a cross-compilation system, the **host machine** is the machine on which the link editor is running, and the **target machine** is the machine on which the output object file will run. For instance, the b16 link editor will run on the PDP-11/70, VAX or 3B20S machines, but the

object file will run only on the target machine for the b16 – the Intel 8086.

On a native A/UX system, the host and the target are normally the same. That is, the link editor on a Macintosh II produces an object file that is executable on that machine.

1.1.2 Memory configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into "configured memory" and "unconfigured memory." **Configured memory** indicates a range of memory for which the appropriate chips have been installed and are available for use. **Unconfigured memory** denotes a range of memory for which no chips have been installed, or that is unavailable for use. The default is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K-1 are then not configured. Unconfigured memory is treated as reserved and is unusable by `ld`.

Note: Nothing may ever be linked 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 linking process. Memory configurations other than the default must be specified explicitly.

Unless otherwise specified, all discussion in this document of memory, addresses, and so on, is about the configured sections of the address space.

1.1.3 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 may be **holes** or gaps

between input sections (and between output sections), storage is allocated contiguously within each output section and may not overlap a hole in memory.

1.1.4 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.

1.1.5 Binding

Often you may 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 may also bind global symbols with an assignment statement in the `ld` command language.

1.1.6 Object files

Object files are produced both by the assembler (typically as a result of calling the compiler) and by `ld`. `ld` accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to `ld` may also be absolute files (see "Nonrelocatable input files" for details).

Files produced by the compiler or assembler always contain three sections

- `.text` containing the instruction text (for example, executable instructions)
- `.data` containing initialized data variables
- `.bss` containing uninitialized data variables

For example, if a C program contained the following global (not inside a function) declarations:

```
int i = 100;
char abc[200];
```

and the following 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 the rule: both initialized and uninitialized statics are allocated to the `.data` section (the value of an uninitialized static in a `.data` section is zero).

1.2 Options

You may 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 just a collection of object files. Thus, as is the case with any object file, libraries are searched as they are encountered. The `L` specifies an alternative directory for searching for libraries. Therefore, to be effective, a `-L` option must 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). The following options are supported:

`-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. This option 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.

`-m` Produces a map or listing 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* may be a full or partial A/UX pathname.

`-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.

`-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, if you use `-s`, you may not use `-r`. All symbols are stripped, including global and undefined symbols.

`-t` Disables checking 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's symbol table. The argument *sym* is the name of the symbol. This is useful for linking entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the linking 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 zero." This option is overridden if any section or memory directives are used.

`-F` Performs alignment necessary for demand paging. Sections will be aligned on stricter boundaries in the address space. Sections will be blocked in the output file so that they begin on file system block boundaries. Also, the magic number 0413 will be 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 as 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 will immediately follow the text section when loaded and stores the magic number 0407 in the header. This prevents the text from being shared (shared text 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` invoked.

`-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 have to 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 arguments' destination, 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.

2. The `ld` command language

2.1 Expressions

Expressions may 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 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 `long ints`.

Symbol names may contain upper or lowercase letters, digits, and the underscore (_). Symbols within an expression have the value of the address of the symbol only. `ld` does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, and so on.

`ld` uses a `lex`-generated input scanner to identify symbols, numbers, operators, and so forth. The current scanner design makes the following names reserved and unavailable as symbol or section names:

```
ALIGN     DSECT    MEMORY       PHY    SPARE
ASSIGN    GROUP    NOLOAD     RANGE       TV
BLOCK    LENGTH    ORIGIN   SECTIONS

align    group    length    origin        spare
assign       l         o       phy
block      len       org     range
```

The operators that are supported are shown in order of precedence in Table 14-1:

Table 14-1. Precedence of operators

Symbols and Functions
! ~ - (unary minus)
* / %
+ - (binary minus)
>> <<
== != > < <= >=
&
|
&&
||
= += -= *= /=

These operators have the same meaning as in the C language. Operators on the same line have the same precedence.

2.2 Assignment statements

External symbols may be defined and assigned addresses via 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 has been performed. This occurs after all input-file-defined symbols are appropriately relocated, but before the actual relocation of the text and

data itself. 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 "Section Definition Directive" under "The `ld` Command Language"). There is a special symbol, "dot" (`.`), however, that may occur only within a section-definition directive. This symbol refers to the current address of `ld`'s location counter. Thus, assignment expressions involving `.` are evaluated during the allocation phase of `ld`.

Assigning a value to the dot (`.`) symbol within a section-definition directive will increment or reset `ld`'s location counter and may create holes within the section (as described in "Section Definition Directives").

Assigning the value of the `.` symbol to a conventional symbol permits the final allocated address of a particular point within the link edit 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's 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.

2.3 Specifying a memory configuration

MEMORY directives are used to specify:

- the total size of the virtual space of the target machine
- the configured and unconfigured areas of the virtual space

If you do not supply any directives, `ld` assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

Using MEMORY directives, you may assign an arbitrary name of up to eight characters to a virtual address range. Output sections then may be forced to be bound to virtual addresses within specifically-named memory areas. Memory names may contain upper or lowercase letters, digits and the special characters `$`, `.` or `_`. Names of memory ranges are used by `ld` only 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. Unconfigured memory is not used in `ld`'s allocation process, and hence nothing may be link edited, bound, or assigned to an address within unconfigured memory.

As an option on the MEMORY directive, you may associate **attributes** with a named memory area. This restricts the memory areas (with specific attributes) to which an output section may 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

`R` readable memory

W	writable memory
X	executable (instructions may reside in this memory)
I	initializable (stack areas are typically not initialized)

Other attributes may be added in the future if necessary. If you do not specify any attributes on 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-lgth
 ...
}
```

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, may 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 linked to the first 0x10000 words of memory, you may do so with a MEMORY directive:

```
MEMORY
{
  valid : org = 0x10000, len = 0xFE0000
}
```

2.4 Region directives

This implementation does not support region specifications.

2.5 Section definition directives

You may 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.

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 linked, each containing the three sections `.text`, `.data`, and `.bss`, the output object file will also contain three sections, `.text`, `.data`, and `.bss`. If two object files are linked, one containing sections *s1* and *s2*, the other containing sections *s3* and *s4*, the output object file will contain 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.

2.5.1 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, as are the file specifications themselves.

If a filename appears with no sections listed, then all sections from the file are linked 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 by

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 there any additional input files that contain input sections named *outsec1*, these sections are linked following the last section named in the *outsec1* definition. If there are any other input sections in *file1*.o or *file3*.o, they will be placed in output sections with the same names as the input sections.

2.5.2 Loading a section at a specified address

You may bond an output section to a specific virtual address, 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 may 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 each section's size 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 4. 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's contents either are accessed incorrectly or are not executed properly. If you specify an odd byte boundary, `ld` will issue a warning message.

2.5.3 Aligning an output section

You may 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)
```

For example,

```
SECTIONS
{
        outsec  ALIGN(0x20000) :
        {
            file-spec  (secname)
        }
        ...
}
```

The output section *outsec* is not bound to any given address, but is linked 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 M68000 systems is to align the code (`.text`) and data (`.data` and `.bss` combined) separately on 512-byte boundaries. Since MMU requirements vary from system to system, alignment is not always desirable. The version of `ld` for M68020 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 section alignment action for `ld` on MC68020 systems is to align the code (`.text`) at byte 0 and the data (`.data` and `.bss` combined) at the 4 megabyte boundary (byte 10487576).

The default allocation algorithm for `ld` is

1. Link all input `.text` sections together into one output section. This output section is called `.text` and is bound to an address of 0x0.
2. Link 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. Link 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 have been processed (and if no override is provided), `ld` will search 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 below. If it does not exist, the default alignment action will be taken.

The `default.ld` file should appear as in the example below, 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      : { }
     }
}
```

where *align-value* is a machine-dependent constant.

Note: The current (MC68020) system requires a *data rounding* of 2 megabytes. This 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 wish 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 could 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); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

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, 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.

2.5.4 Creating holes within output sections

The special symbol dot (`.`) appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, `.` causes `ld`'s 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 "Options" under "`ld`: The Link Editor" and the discussion of filling holes in "Initialized Section Holes or `.bss` Sections" below.

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 linked 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 *f7*`.o` is linked 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. As a result, if, in the above example, *outsec* ultimately is linked to start at an odd address, the part of *outsec* built from *f7*`.o(.text)` also starts at an odd address, even though *f7*`.o(.text)` is aligned to a full word boundary. You may prevent this by specifying an alignment factor for the entire output section:

```
outsec ALIGN(4) : {
```

You should note that the assembler, `as`, always pads the sections it generates to a full word length, making explicit alignment specifications unnecessary. This also holds true for the compiler.

Expressions that decrement `.` 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 . are += and align.

2.5.5 Creating and defining symbols at link-edit time

You may use the assignment instruction of ld to give symbols a value that is link-edit-dependent. Typically, there are three types of assignments:

1. Use of . to adjust ld's location counter during allocation
2. Use of . to assign an allocation-dependent value to a symbol
3. Assigning an allocation-independent value to a symbol

The first case has already been discussed in the previous section.

The second case provides a means to assign addresses (known only after allocation) to symbols; for example,

```
SECTIONS
{
     outsc1 : {file-spec  (secname) }
     outsc2 :
     {
          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
{
     outscl :
     {
          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 *file1*`.o`'s `.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 `.` must appear within `SECTIONS` definitions, since they are evaluated during allocation. Assignment instructions that do not involve `.` may 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 referenced within a set of object files being link-edited, 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.

2.5.6 Allocating a section into named memory

You may specify a section to be linked somewhere within a specific, named memory (as previously specified on a `MEMORY` directive) (the > notation is borrowed from the UNIX system concept of "redirected output").

For 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 directs `ld` to place *outsec1* anywhere within the memory area named *mem1* (somewhere within the address range `0x0-0xFFFF` or `0x120000-0x123FF`). The *outsec2* is to be placed somewhere in the address range `0x70000-0xAFFFF`.

2.5.7 Initialized section holes or `.bss` sections

When holes are created within a section (as in the example in "Creating Holes Within Output Sections"), `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, nor are they supplied by the link editor.

You may use initialization options in a `SECTIONS` directive to set such holes or to set `.bss` sections as output to an arbitrary 2-byte pattern.

> *Note:* Such initialization options apply only to `.bss` sections or holes.

As an example, in an application 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 may specify 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, if part of such a section is initialized, the entire 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 will hold:

- Explicit initialization options must be used to initialize all `.bss` sections in the output section.
- `ld` will use 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:
     {
          f7.o (.bss)
          ...
     } = 0xFFFF
     sec4: {f4.o (.bss) }
}
```

In the example above, 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.

3. Notes and special considerations

3.1 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`

Archive libraries are created through the use of the A/UX system `ar` command from object files generated by running `cc` or `as`.

An archive library is always processed using **selective inclusion:** only those members that resolve existing undefined-symbol references are taken from the library for link editing.

Libraries may be placed both inside and outside section definitions. In both cases, a member of a library is included for linking 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 into `.data`, the `.bss` section of the member into `.bss`, and so on. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that:

- Specific members of a library may not be referenced explicitly in an i-file.
- The default rules for the placement of members and sections may not 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 may 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, for a member to be extracted from the library, it must satisfy a reference that is known to be unresolved at the time the library is searched.

You may 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` will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

`ld`, running on the Macintosh II, uses a random access library. All machines running a pre-V.0 UNIX system use an old format library that must be searched linearly.

The old format library is in use on all machines running a pre-V.0 UNIX system.

The link editor will make one search through a library in the old format, but will continue to search through a library in the new format until it has determined that it can resolve no more references from that library. Because of the different searching algorithms used, programs that are link edited on machines with different archive formats and are otherwise the same may include files from libraries in a different order.

Be careful when using archive libraries in a subsystem loading environment. For a member of an archive (an object file) to be included in a subsystem final load file, there must be a reference within the subsystem being linked to a symbol defined in that object file. You may use the `-u` option to create unresolved references that will 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 linking. 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 may use the -u option to force the linking of library members when the link edit 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's global symbol table. If any member of library liba.a defines this symbol, it, and perhaps other members as well, is extracted. Without the -u option, there would have been no trigger to cause ld to search the archive

library.

3.2 Dealing with holes in physical memory

When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user has the responsibility of forming output sections that will 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 may be placed. Appropriate directives must be supplied to break up the `.text` output section so `ld` may do allocation. For example,

```
SECTIONS
{
     txt1 :
     {
          f1.o (.text)
          f2.o (.text)
          f3.o (.text)
     }
     txt2 :
     {
          f4.o (.text)
          f5.o (.text)
          f6.o (.text)
     }
     ...
}
```

3.3 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 may be made up of zero or more input sections. After an output section's composition 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 link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Allocate any output sections for which explicit bonding addresses were specified.
2. Allocate any output sections to be included in a specific named memory. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment taken into consideration.
3. Allocate output sections that are not handled by one of the above steps.

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 they fit.

3.4 Incremental link editing

As previously mentioned, the output of `ld` may be used as an input file to subsequent `ld` runs, providing that the relocation information is retained (`-r` option). With large applications you may find it desirable to partition C programs into subsystems, link each subsystem independently, and then link edit 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 may achieve a form of **incremental link editing**, whereby it is necessary to relink only a portion of the total link edit when a few programs are recompiled.

To apply this technique, there are two simple rules:

1. Intermediate link edits should contain only `SECTIONS` declarations and be concerned only with the formation of output

sections from input files and input sections. You should not do any binding of output sections in these runs.

2. All allocation and memory directives, as well as any assignment statements, are included in the final `ld` call only.

3.5 DSECT, COPY, and NOLOAD sections

You may give sections a type in a section definition, as shown in the following example:

```
SECTIONS
{
    name1 0x200000 (DSECT)    : {file1.o}
    name2 0x400000 (COPY)     : {file2.o}
    name3 0x600000 (NOLOAD)   : {file3.o}
}
```

The `DSECT` option creates what is called a "dummy section." A **dummy section** has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map (the `-m` option) generated by `ld`.
2. It may overlay other output sections and even unconfigured memory. `DSECT`s may overlay other `DSECT`s.
3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the `DSECT` were actually loaded at its virtual address. Other input sections may reference `DSECT`-defined symbols. Undefined external symbols found within a `DSECT` cause specified archive libraries to be searched; any members that define such symbols are link edited normally (not in the `DSECT` or as a `DSECT`).
4. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from *file1*`.o` are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global

symbols and they are resolved correctly.

Something called a "copy section" is created by the COPY option. This is similar to a dummy section. 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 section of the type NOLOAD differs in only one respect from a normal output section: text and data are not written to the output file.

A **noload section** is allocated virtual space, appears in the memory map, and so forth.

3.6 Output file blocking

You may 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 *pfiles* for DMERT. The output sections of a pfile have certain requirements in terms of physical file offsets. These requirements may be met using BLOCK and -B.

You may 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 nor on any part of the link edit 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 assures 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).

3.7 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 may be removed by `ld` (see the `-s` option in "Options" under "`ld`: The Link Editor") or by the `strip`(1) program. Note, however, that the link edit run continues, using the nonrelocatable input file. For such a link edit to be successful (that is, actually and correctly to link edit all input files, relocate all symbols, resolve unresolved references, and so on), two conditions on the nonrelocatable input files must be met:

1. Each input file must have no 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, you must take extreme care when supplying such input files to `ld`.

3.8 The `-ild` option

When the `-ild` option is used, the link editor creates a pair of dummy sections, `DSECT`s, 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 to 99. At most, 50 pairs of these sections will be 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.

4. Error messages

4.1 Corrupt input files

Certain error messages indicate that the input file is corrupt, nonexistent, or unreadable. If you get 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. These error messages include

`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.

4.2 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. Messages to this effect include

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*.

4.3 Internal errors

Certain messages indicate that something is wrong with `ld` internally. If you get them, there is probably nothing you can do except to get help from another experienced user of `ld`. Such messages include

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.`

4.4 Allocation errors

Certain error messages appear during the allocation phase of the link edit. 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 get 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 `ld` Command Language" and "Notes and Special Considerations." These messages include

`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.

4.5 Misuse of link editor directives

Certain error messages are explanations that occur following the misuse of an input directive. If you get them, please review the appropriate section in the manual. These messages include

Adding *name* (*sect*) to multiple output sections.
: The input section is mentioned twice in the SECTIONS directive.

Bad attribute value in MEMORY directive: *c*.
: 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 of a SECTIONS directive.

Bad fill value.
: The fill value must be a 2-byte constant.

`Bonding excludes alignment.`
: The section will be 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 may be aligned or bound to an address, but the sections making up the group may not 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.

`Regions commands not allowed`
: The A/UX 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 `ld` does not accept all possible commands.

4.6 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 the manual.

`Absolute symbol` *`name`* `being redefined.`
: An absolute symbol may not be redefined.

`ALIGN illegal in this context.`
: Alignment of a symbol may only be done 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 may not use the dot symbol (`.`) in assignment statements that are outside of `SECTIONS` directives.

`Symbol` *name* `is undefined.`
All symbols referenced in an assignment statement must be defined.

`Symbol` *name* `from file` *name* `being redefined.`
A defined symbol may not be redefined in an assignment statement.

`Undefined symbol in expression.`
All symbols used in expressions must be defined.

4.7 Misuse of options

Certain errors arise from the misuse of options. If you get any of the following messages, please review the appropriate section of the manual:

`Both -r and -s flags are set.`
`-s flag turned off.`
Further relocation requires a symbol table.

`Can't find library lib`*x*`.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 white space before the argument, some do not; see "Options." Including extra white space or not including the required white 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*.

4.8 Space constraints

Certain error messages may occur if `ld` attempts to allocate more space than is available. If you get them, you should attempt to decrease the amount of space used by `ld`. You may do this by making the i-file less complicated or by using the `-r` option to create intermediate files. These space-constraint messages include

`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.`

4.9 Miscellaneous errors

Errors occur for many reasons. If one occurs that has not been explained in a previous section, refer to the error message for an indication of where to look in the manual. Miscellaneous error messages include

`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, the link editor requires that all archives have a symbol table. This symbol table may 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. This would 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 may 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*.
: Ifiles may 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 may not have been 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 will happen 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 is generally caused by an interaction of `yacc`(1) and `cc`(1). See "Notes and Special Considerations."

`Undefined symbol` *sym* `first referenced in file` *name*. Unless you use the `-r` option, the `ld` requires that all

referenced symbols are defined.

`Unexpected EOF (End Of File).`
Syntax error in the i-file.

5. Syntax diagram for input directives

The following tables contain syntax diagrams for input directives. For *flags*, wherever there is a space between a flag option and its argument, one or more blanks, tabs, or newlines may be substituted.

Note: Number suffixes have been added to some metalanguage terms to illustrate treatment of multiple arguments. These suffixes should be ignored when seeking the definition of such terms.

Directive	→	Expanded directive
file	→	*cmd* ...
cmd	→	*memory*
	→	*sections*
	→	*assignment*
	→	*filename*
	→	*flags*
memory	→	`MEMORY { ` *memory-spec*
		`[ [ , ]` *memory-spec* `}`
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* ... `}`

Directive	→	Expanded directive
sec-or-group	→	*section*
	→	*group*
	→	*library*
group	→	GROUP *group_options* : { *section-list* } [*mem-spec*]
section-list	→	*section1* [[,] *section2*] ...
section	→	*name sec-options* : { *statement-list* } [*fill*] [*mem-spec*]
group-options	→	[*addr*] [*align-option*]
sec-options	→	[*addr*] [*align-option*] [*block-option*] [*type-option*]
addr	→	*long*
align-option	→	*align* (*long*)
align	→	ALIGN
	→	align
block-option	→	*block* (*long*)
block	→	BLOCK
	→	block

Directive	→	Expanded directive
type-option	→	`(DSECT)`
	→	`(NOLOAD)`
	→	`(COPY)`
fill	→	= *long*
mem-spec	→	> *name*
	→	> *attributes*
statement	→	*filename* [(*name-list*)] [*fill*] *library assignment*
statement-list	→	*statement1* [*statement2*] ...
name-list	→	*name* [[,] *name*] ...
library	→	`-l`*name*
assignment	→	*lside assign-op expr end*
lside	→	*name*
	→	.
assign-op	→	=
	→	+=
	→	-=
	→	*=
	→	/=
end	→	;
	→	,

Directive	→	Expanded directive
expr	→	*expr binary-op expr*
	→	*term*
binary-op	→	*
	→	/
	→	%
	→	+
	→	−
	→	>>
	→	<<
	→	==
	→	!=
	→	>
	→	<
	→	<=
	→	>=
	→	&
	→	|
	→	&&
	→	||
term	→	*long*
	→	*name*
	→	*align* (*term*)
	→	(*expr*)
	→	*unary-op term*
unary-op	→	!
	→	−

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. This may include a full or partial pathname.
pathname	→	Any valid A/UX operating system pathname (full or partial)

Chapter 15
COFF Reference

Contents

Figures

Tables

Chapter 15

COFF Reference

1. COFF: The Common Object File Format

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 COFF's key features are

- Applications may 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 may make some modifications in the object file construction at compile time.

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains:

- 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 15-1 shows the overall structure.

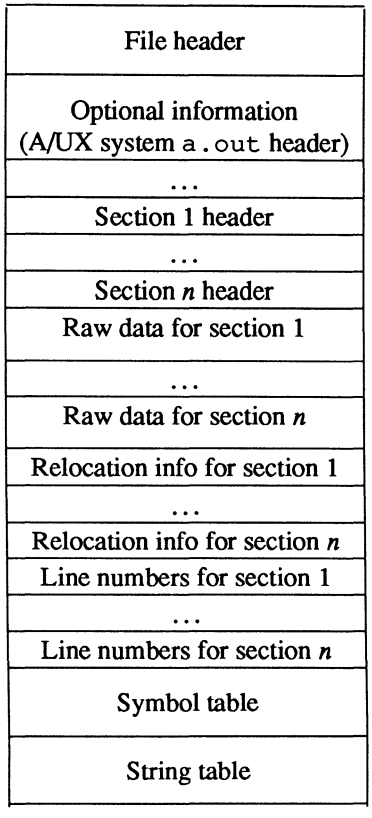


Figure 15-1. Object file format

The last four sections (relocation, line numbers, symbol table, and the string table) may 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's (`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 may 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.
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.

2. File Header

The file header contains the 20 bytes of information shown in the following table. 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 15-1. 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.

2.1 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`.

2.2 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 15-2.

Table 15-2. 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 has had its bytes **swabbed** (that is, the bytes of symbol table name entries have been 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

where 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).

2.3 File header declaration

The C structure declaration for the file header is given in Figure 15-2. You may find this declaration in the header file `filehdr.h`. See `filehdr`(4) in *A/UX Programmer's Reference*.

```
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)
```

Figure 15-2. File header declaration

3. 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`.

3.1 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 15-3.

Table 15-3. 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 that machine's operating system how that file should be executed. The magic numbers recognized by the A/UX operating system are shown in Table 15-4.

Table 15-4. 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.

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*.

3.2 Optional header declaration

The C language structure declaration used for the A/UX system `a.out` file header is given in Figure 15-3. This declaration may be found in the header file `aouthdr.h`.

```
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;
```

Figure 15-3. `aouthdr` declaration

4. 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 15-5.

Table 15-5. 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 may be used to locate the start of data, relocation, or line number entries for the section. They may be readily used with the A/UX operating system function `fseek`(3S).

4.1 Flags

The lower 4 bits of the flag field indicate a section type as shown in Table 15-6.

Table 15-6. 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	Noload 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
STYP_BSS	0x80	Section contains only uninitialized data

4.2 Section header declaration

The C structure declaration for the section headers is described in Figure 15-4. You can find this declaration in the header file `scnhdr.h` (see `scnhdr`(4) in *A/UX Programmer's Reference*):

```
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)
```

Figure 15-4. Section header declaration

4.3 `.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.

5. Sections

Section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begin 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 14, "`ld` Reference") let you

- 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 together (each containing the three sections `.text`, `.data`, and `.bss`), the output object file will also contain those three sections.

6. 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 15-7.

Table 15-7. 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 4 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 0, 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 15-8, and are documented in the header file `reloc.h`.

Table 15-8. VAX and M68000 relocation types

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_RELBYTE	017	Direct 8-bit reference to the symbol's virtual address.
R_RELWORD	020	Direct 16-bit reference to the symbol's virtual address.
R_RELLONG	021	Direct 32-bit reference to the symbol's virtual address. (a VAX relocation type)
R_PCRBYTE	022	A PC-relative 8-bit reference to the symbol's virtual address.
R_PCRWORD	023	A PC-relative 16-bit reference to the symbol's virtual address.
R_PCRLONG	024	A PC-relative 32-bit reference to the symbol's 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's start address and the program's 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.

6.1 Relocation entry declaration

The structure declaration for relocation entries is given in Figure 15-5. This declaration can be found in the header file `reloc.h`.

```
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) */
```

Figure 15-5. Relocation entry declaration

7. 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 breakpoint can be inserted. You can then reference line numbers using a software debugger like `sdb`. All line numbers in a section are grouped by function as shown in Figure 15-6.

Symbol index	0
Physical address	Line number
Physical address	Line number
...	
Symbol index	0
Physical address	Line number
Physical address	Line number

Figure 15-6. Line number grouping

The first entry in a function grouping has line number 0 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.

7.1 Line number declaration

Figure 15-7 contains the structure declaration currently used for line number entries. This declaration can be found in the header file `linenum.h`.

```
struct 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) */
```

Figure 15-7. Line number entry declaration

8. 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 15-8.

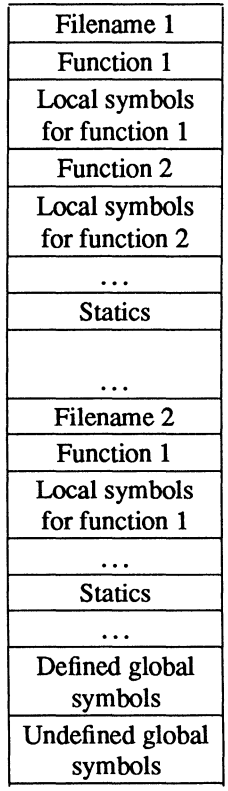


Figure 15-8. COFF global symbol table

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.

8.1 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 15-9.

Table 15-9. Special symbols in the symbol table

Symbol	Meaning
`.file`	Filename
`.text`	Address of `.text` section
`.data`	Address of `.data` section
`.bss`	Address of `.bss` section
`.bb`	Address of start of inner block
`.eb`	Address of end of inner block
`.bf`	Address of start of function
`.ef`	Address of end of function
`.target`	Pointer to the structure or union returned by a function
`.xfake`	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 will be .0fake , .1fake , and .2fake .

Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entry.

8.2 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 15-9 shows this sequence:

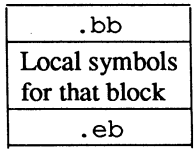


Figure 15-9. Special symbols

Because inner blocks may be nested by several levels, the .bb/.eb pairs and associated symbols may also be nested. The code illustrated in Figure 15-10 is used as an example of 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 15-10. Nested blocks

The symbol table built for the coding example in Figure 15-10 is shown in Figure 15-11.

.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: i
.eb for block 4
.eb for block 1

Figure 15-11. Example of the symbol table

8.3 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 Figure 15-12.

Function name
.bf
Local symbol
.ef

Figure 15-12. Symbols for functions

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 in Figure 15-13.

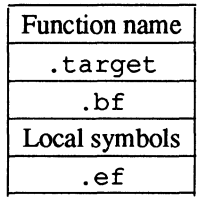


Figure 15-13. The special symbol `.target`

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 0.

8.4 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 15-10. 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 15-10. 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 8 bytes in the symbol table entry are the symbol name field. This field is defined as the union of a character array and two `longs`. A symbol name may 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 8 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 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes, as shown in Table 15-11.

Table 15-11. Name field

Bytes	Declaration	Name	Description
0-7	`char`	`n_name`	8-character null-padded symbol name
0-3	`long`	`n_zeroes`	Zero 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". 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 Tables 15-12 and 15-13. You can find these `defines` in the header file `storclass.h`.

Table 15-12. Storage classes (page 1 of 2)

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
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

Table 15-13. Storage classes (page 2 of 2)

Mnemonic	Value	Storage class
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

```
C_EFCN
C_EXTDEF
C_ULABEL
C_USTATIC
C_LINE
```

Some special symbols are restricted to certain storage classes, listed in Table 15-14.

Some storage classes are used only for certain special symbols as shown in Table 15-15.

Table 15-14. 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
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Table 15-15. Restricted storage classes

Storage class	Special symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

The meaning of a symbol's value depends on its storage class. This relationship is summarized in Tables 15-16 and 15-17.

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 15-16. Storage class and value (page 1 of 2)

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

Table 15-17. Storage class and value (page 2 of 2)

Storage class	Meaning
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_FILE	(See text)

Section numbers are declared in the header file `syms.h` and are listed in Table 15-18:

Table 15-18. 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, `typedefs`, 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 0 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 0 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 0 and a nonzero value.

Symbols having certain storage classes are also restricted to certain section numbers. They are shown in Tables 15-19 and 15-20.

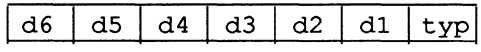
Table 15-19. Section number and storage class (page 1 of 2)

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

Table 15-20. Section number and storage class (page 2 of 2)

Storage class	Section number
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 M68020 `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



Bits 0 through 3, called `typ`, indicate one of the fundamental types given in Table 15-21.

Table 15-21. 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_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short
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 given in Table 15-22.

Table 15-22. 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 the 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."

Tables 15-23 and 15-24 show the type entries that are legal for each storage class.

Table 15-23. Type entries by storage class (page 1 of 2)

Storage class	d entry			typ entry basic type
	Function	Array	Pointer	
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

Table 15-24. Type entries by storage class (page 2 of 2)

Storage class	d entry			typ entry basic type
	Function	Array	Pointer	
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

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 *array*s, they are changed to *pointer*s by default. Therefore, no *function* argument can have *array* as its first derived type.

The C language structure declaration for the symbol table entry is given in Figure 15-14. This declaration can be found in the header file syms.h.

```
struct syment {
   union {
      char _n_name[SYMNMLEN]; /* symbol name*/
      struct {
         long _n_zeroes;     /* 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_zeroes   _n._n_n._n_zeroes
#define n_offset   _n._n_n._n_offset

#define SYMNMLEN   8
#define SYMENT     struct syment
#define SYMESZ     18     /* symbol table entry size */
```

Figure 15-14. Symbol table entry declaration

8.5 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 15-25 lists auxiliary table entry formats by type and storage class:

Table 15-25. Auxiliary symbol table entries

Name	Storage class	Type entry		Auxiliary entry format
		d2	typ	
.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 0 through 13. The remaining bytes are 0, regardless of the size of the entry.

The auxiliary table entries for sections have the format as shown in Table 15-26.

Table 15-26. 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 15-27.

The auxiliary table entries for the end of structures have the format shown in Table 15-28.

The auxiliary table entries for functions have the format shown in Table 15-29.

The auxiliary table entries for arrays have the format shown in Table 15-30.

The auxiliary table entries for the beginning of blocks have the format shown in Table 15-31.

The auxiliary table entries for the end of blocks have the format shown in Table 15-32.

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Table 15-33.

Table 15-27. 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 15-28. 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 15-29. 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's address in the transfer vector table (not used by A/UX operating system)

Table 15-30. 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 15-31. 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 15-32. 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 15-33. 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` may or may 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 Figures 15-15 and 15-16. This declaration may be found in the header file `syms.h`.

```
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;
```

Figure 15-15. Auxiliary symbol table entry (page 1 of 2)

```
  } x_sym;
    struct {
       char x_fname[FILNMLEN];
    } x_file;
    struct {
       long           x_scnlen;
       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
```

Figure 15-16. Auxiliary symbol table entry (page 2 of 2)

9. 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 4.

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 15-34.

Table 15-34. String table

28			
'1'	'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.

10. 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 may 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

Advanced UNIX Programming
Marc J. Rochkind
Prentice-Hall, 1985
{UNIX system calls}

C: A Reference Manual
Samuel P. Harbison, Guy L. Steele, Jr.
Prentice-Hall, 1984

MC68020 32-Bit Microprocessor User's Manual, 2nd. ed.
Motorola, 1985

MC68881 Floating-Point Coprocessor User's Manual
Motorola, 1985

System V Interface Definition
AT&T, 1986

The C Programming Language
Brian W. Kernighan, Dennis M. Ritchie
Prentice-Hall, 1978

The Design of the UNIX Operating System
Maurice J. Bach
Prentice-Hall, 1986
{internal algorithms and data structures}

The Elements of Programming Style
Brian W. Kernighan, P. J. Plauger
McGraw-Hill, 1974
{coding and design techniques}