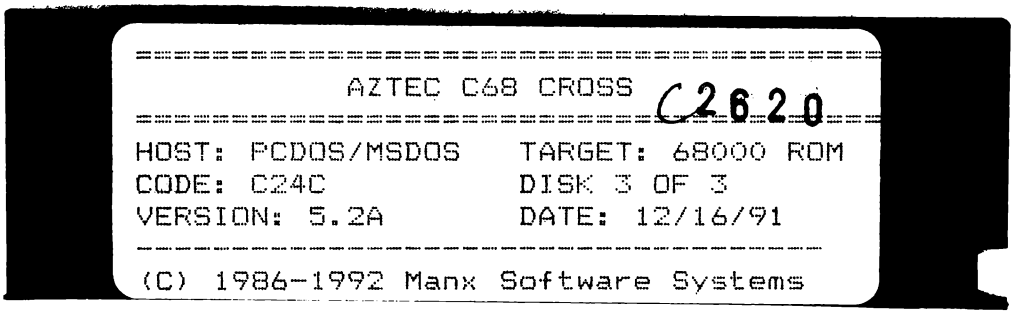
Aztec C68k/ROM

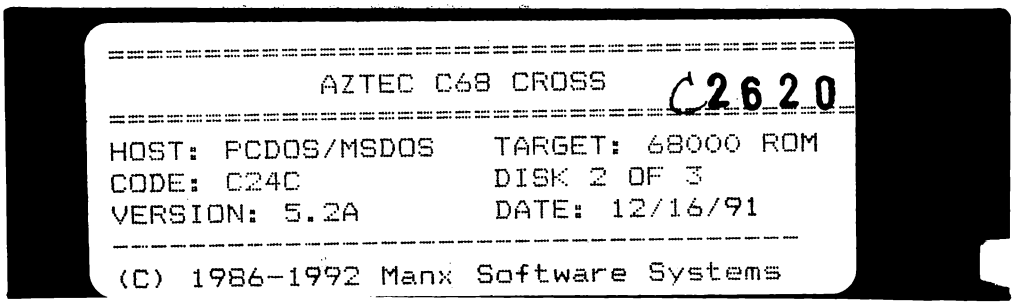
68000 Family
ROM Development Software
for MS DOS/PC DOS Host Systems

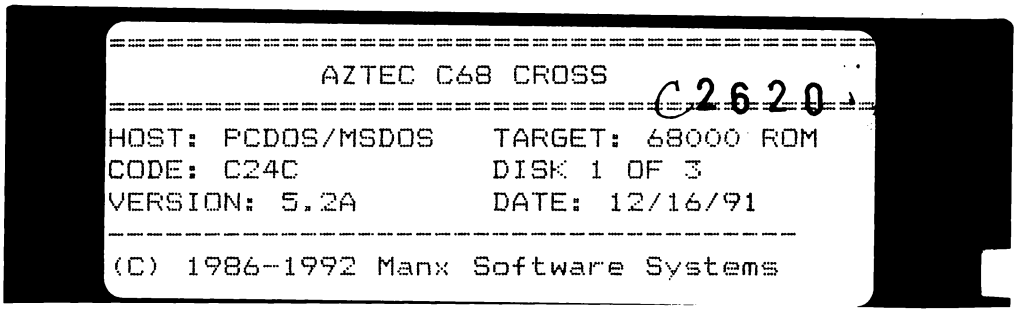
Version 5.0
November 1991

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(908) 542-2121


AZTEC C68 CROSS
C2620
HOST: PCDOS/MSDOS
TARGET: 68000 ROM
CODE: C24C
DISK 3 OF 3
VERSION: 5.2A
DATE: 12/16/91
(C) 1986-1992 Manx Software Systems


AZTEC C68 CROSS
C2620
HOST: PCDOS/MSDOS
TARGET: 68000 ROM
CODE: C24C
DISK 2 OF 3
VERSION: 5.2A
DATE: 12/16/91
(C) 1986-1992 Manx Software Systems


AZTEC C68 CROSS
C2620
HOST: PCDOS/MSDOS
TARGET: 68000 ROM
CODE: C24C
DISK 1 OF 3
VERSION: 5.2A
DATE: 12/16/91
(C) 1986-1992 Manx Software Systems

USE RESTRICTIONS

You are permitted to install and use this product on a single computer. Multiple CPU systems require supplementary licenses.

Before using any Aztec C products, the License Registration included with this product must be signed and mailed to:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702

COPYRIGHT

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to this product and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to modify the programs and revise the contents of the manual without obligation to notify any person of such revision or changes.

TRADEMARKS

Aztec C, Manx AS, Manx LN, Manx Host, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MS-DOS is a trademark of Microsoft. PC-DOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh, Apple, and Apple II are trademarks of Apple Computer. Atari is a trademark of Atari Computers. Amiga is a trademark of Commodore-Amiga.

Manual Revision History

November 1991	Third Edition
November 1987	Second Edition
January 1987	First Edition

Suggestions for Further Reading

Manx Software Systems recommends the following books as additions to your C development library. The books listed here will serve as helpful supplements to your standard Aztec C documentation.

Most of these books are available directly from Manx Software Systems. To order or for more information contact our Sales Department at 1-800-221-0440. (International call 908-542-2121.) Or, you may write to us at: Manx Software Systems, P.O. Box 55, Shrewsbury, NJ 07702.

- *The C Programming Language* Second Edition by Brian W. Kernighan and Dennis M. Ritchie. (Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Programming in C* by Stephen G. Kochan. (Hayden Book Company, Hasbrouck Heights, New Jersey 07604)

- *C: A Reference Manual* Third Edition by Samuel P. Harbison and Guy L. Steele, Jr. (Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Motorola's M68000 16/32-bit Microprocessor Programmer's Reference Manual.* (Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Motorola's MC68851 Paged Memory Management Unit User's Manual.*(Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Motorola's MC68030 Enhanced 32-Bit Microprocessor User's Manual.*(Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Motorola's MC68881/882 Floating-Point Coprocessor Manual.*(Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Motorola's MC68020 32-Bit Microprocessor User's Manual.*(Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Motorola's MC68000 16-Bit Microprocessor User's Manual.*(Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)

- *Programming the 68000* by Steve Williams. (SYBEX, Inc., 2344 Sixth Street, Berkeley, California 94710)

- *From Chips To Systems* by Rodnay Zaks and Alexander Wolfe. (SYBEX, Inc., 2344 Sixth Street, Berkeley, California 94710)

- *68000, 68010, 68020 Primer* by Stan-Kelly-Bootle and Bob Fowler. (Howard W. Sams and Company, 4300 West 62nd Street, Indianapolis, Indiana 46268)

Of course, there are many other useful books on the market available for C programmers. This list has been included to help you get started.

Table of Contents

Chapter 1 - Overview

Chapter 2 - ROM Tutorial

Chapter 3 - Compiler

Chapter 4 - Assembler

Chapter 5 - Linker

Chapter 6 - Utilities

Chapter 7 - Z Editor

Chapter 8 - Library Customization

Chapter 9 - Library Overview

Chapter 10 - Library Functions

Chapter 11 - Technical Information

Chapter 12 - Error Messages

Index

OVERVIEW 1

Chapter 1 - Overview

Aztec C68k/ROM is a set of programs for developing programs in the C programming language; the resulting programs run on ROM- and/or RAM-based systems that use a Motorola 68000-family microprocessor.

The Aztec C68k/ROM is a comprehensive ROM development system including the following features:

- The full C language, as defined by the American National Standards Institute (ANSI) is supported.
- An extensive set of user-callable functions is provided, in both source and object form. You will probably need to modify some of the system dependent functions. For example, you will probably need to customize the startup routine; and if you want to use either the standard I/O or unbuffered I/O functions, you will need to write the unbuffered I/O functions.
- Modular programming is supported, allowing the components of a program to be compiled separately, and then linked together.
- Assembly language code can either be combined in-line with C source code, or placed in separate modules which are then linked with C modules.

- Programs can be generated in several formats, including Motorola S-records, Intel Hex records, and Tektronix TekHex records. ROM chips generated from these records will contain the program's code and a copy of its initialized data.

- A ROM program can contain both initialized and uninitialized global and static variables. When the program starts, its initialized variables in RAM will be automatically set from the copy in ROM, and its uninitialized variables will be cleared.

The functions provided with this package are ANSI compatible and are compatible with Aztec C packages provided for other systems. Thus, once you have customized the functions, you can create programs that will run on ANSI-based systems or on other systems supported by Aztec C with little or no change.

Components

Aztec C68k/ROM contains the following components:

- **c68**, the Aztec compiler;
- **as68**, the Aztec assembler;
- **ln68**, the Aztec linker;
- **lb68**, the object module librarian;
- **z**, a text editor that is compatible with the UNIX vi editor;
- Source and object code for the library functions;
- **make, grep**, and **diff**, utilities that are compatible with the UNIX programs of the same name.
- Several utility programs

Documentation

The Aztec C68k/ROM chapters included in this manual are:

- **Tutorial** describes how to get started with Aztec C68k/ROM: it discusses the installation of Aztec C68k/ROM and gives an overview of the process for turning a C source program into Motorola S-records, Intel hex records and other supported formats;
- **Compiler, Assembler,** and **Linker** present detailed information on using the compiler, assembler, and linker;
- **Utilities** describes the utility programs that are provided with Aztec C68k/ROM;
- **Z Editor** describes the Z text editor and all of its features;
- **Library Customization** describes (1) modifications you can make to the provided library source; and (2) the creation of object module libraries from the provided source;
- **Library Overview** presents an overview of the functions provided with Aztec C;
- **Library Functions** describes the system-independent functions provided with Aztec C68k/ROM;
- **Technical Information** discusses miscellaneous topics, including C-callable assembly language functions, and C language interrupt handlers;
- **Error Messages** lists and describes the error messages that are generated;
- **Technical Support** details the guidelines to be followed when in need of technical assistance.

Wide Range of Choices

Aztec C host and target packages are available for the following combinations:

Host	Target
PC-DOS/MS-DOS	8086/80186/80286/8087/80287
PC-DOS/MS-DOS	6502/65C02
PC-DOS/MS-DOS	68000/68010/68020/68881/68851
PC-DOS/MS-DOS	8080/8085/Z80/64180/Z180
Macintosh	68000/68010/68020/68881/68851
CP/M-80	8080/Z80

For information on these Aztec C Cross Development Systems, as well as the numerous Aztec C Native Development Systems that are available, contact the Manx Software Systems Sales Department at 1-800-221-0440 (International call 908-542-2121.)

How to Proceed

Now that you have Aztec C, the first thing you are going to do is...

...read the manual?

If you are like most users, probably not. The Manx manuals are a very thorough and very important part of your Aztec C package, but we recognize that you may be anxious to put Aztec C to work as soon as you can. To some users, that means "skimming" the important parts and getting to the rest later.

In light of this, we would like to offer a bit of advice. The list below, for both new and experienced users, shows how we suggest you proceed. This may help you off to a quicker start.

- ❑ Step 1: Read the README file (on disk).
- ❑ Step 2: Install your Aztec C disks. Installation instructions have been included in the **Tutorial** chapter.
- ❑ Step 3: Read the remainder of the **Tutorial** chapter which takes you through a step by step process of how to produce a ROMable application.
- ❑ Step 4: Read through **Library Customization** Chapter which discusses how to implement the low-level functions necessary in order to use any additional functions documented in **Library Functions** chapter
- ❑ Step 5: Refer to **Compiler** and **Linker** chapters for correct option use.
- ❑ Step 6: Refer to the **Utilities** chapter for information on the format conversion utilities: Motorola S-rec format (**srec68**), Intel Hex format (**hex68**), and Extended Tek-hex format (**tekhex68**)

Once you have followed the above steps, you should have a better understanding of your Aztec C Cross Development System and its capabilities. You should look through the rest of the documentation, paying particular attention to the **Compiler**, **Assembler**, and **Linker** chapters.

Other chapters that have been affected by the ANSI standard and should be reviewed are **Utilities, Technical Information**, and **Error Messages**.

This manual is straightforward and very thorough, and is designed to be a helpful aid to your programming efforts.

About This User Guide

Throughout this manual, we use the following conventions:

prototype	is used in "prototype statements" to indicate how data should be entered by the user
`input`	to indicate data entered by the user (e.g., commands, options, and functions)
DEFINITION	small uppercase bold is used on terms that may be new to the user; most will probably include explanation or definition of term
{choice1 | choice2}	braces and a vertical bar mean that you have a choice between two or more items
placeholders	information that must be supplied by the user; for example, *filename, range, identifier, etc.*
`output`	to show text that is generated by the computer
<key>	is used in examples to indicate the actual key to press, i.e. **<Enter>, <ESC>**, etc.
^*x*	is used to indicate that you should hold down the control key while typing the specified letter
[*optional*]	to show optional information

Chapter 2 - Tutorial Introduction

This chapter describes how to quickly start using your Aztec C68k/ROM cross development software. It discusses the following topics:

- Installing the Aztec C68k/ROM software on your system.
- Customizing the library functions
- Special features of Aztec C68k/ROM.
- Introduction to the rest of the manual.

Ideally, this chapter should consist of a cookbook set of steps that you can follow to get started using Aztec C68k/ROM. However, since one of those steps, customizing the library, is system dependent, we recommend that you follow the first step, which leads you through the installation of Aztec C68k/ROM on your system, and then simply read the rest of this chapter to get an idea of how programs are developed using Aztec C68k/ROM. Then you can read the **Library Customization** chapter, make any needed revisions to the library function source, and generate your libraries. Finally, you can translate a C program into a ROM-burnable format, by following the "Creating a Program" steps in this chapter.

Installation

To install Aztec C68k/ROM onto your hard disk, place the first distribution disk in a floppy drive and type **install**. The installation program will display a dialog box that lets you to define installation choices. Once you have chosen your options the installation program will automatically complete the installation.

The Destination Directory

When the installation is completed, the Aztec C68k/ROM files will be on your hard disk in the destination directory that you specified to **install**. This directory contains the following subdirectories:

Directory	Contents
BIN	executable programs
INCLUDE	include files
LIB	library functions

The Aztec C68k/ROM Set Up Commands

Before you can proceed to use the installed Aztec C68k/ROM programs, you must execute MS DOS commands that add the BIN directory to the MS DOS search path and set up environment variables that are used by Aztec C68k/ROM programs.

The **install** program has created these commands for you and has written them to the batch file named **az68.bat** in the BIN directory. Thus, to execute these commands you can **cd** into the BIN directory and type:

```
az68
```

You can also add the commands that are in this file to your **autoexec.bat** file so that whenever your system starts it will automatically be prepared to use Aztec C68k/ROM.

Customizing the Libraries

Aztec C68k/ROM provides many library functions. Many of the functions are system independent and some are system dependent.

Before you can use Aztec C68k/ROM, you will probably have to customize some of the system dependent library functions. The **Library Customization** chapter discusses some of the changes that you might want to make to the library; it also describes how to regenerate the libraries after you have customized the library source.

Creating a Program

In this section we will lead you through the steps necessary to translate a C source program named **exmpl.c** into hex code that can be burned into ROM. For a diagram of this procedure, see Figure 2-1. The code for this program will reside in ROM, beginning at memory location **0**. Its data will reside in RAM, beginning at location **08000**.

Step 0: Create the Source Program

The first step to creating a C program is, of course, to create a disk file containing its source. For this, you can use any text editor. We'll assume the source exists, in the file **exmpl.c**.

Steps 1 and 2: Compile and Assemble

To compile and assemble **exmpl.c** enter the following command:

```
c68 exmpl.c
```

This first starts the **c68** compiler, which translates the C source that is in **exmpl.c** into assembly language source. When done, **c68** starts the **as68** assembler. **as68** assembles the assembly language source for the sample program, translating it into object code and writing the object code to the file **exmpl.r** in the current directory. When done, **as68** deletes the file that contains the assembly language source, since it is no longer needed.

There are several compiler options that define a module's characteristics. For this example, we have let these options assume their default values. Later in this chapter we introduce some of these characteristics.

Step 3:Link

The object code version of the **exmpl** program must next be linked to needed functions that are in the **c.lib** library of object modules and converted into a loadable format.

Before linking, make sure that the CLIB68 environment variable is set to the name of the directory that contains the Aztec C68k/ROM object module library.

The command to link the sample program is:

```
ln68 +d 8000 -o exmpl exmpl.r -lc
```

There's several parameters to this command, so let's go through them, one at a time.

Positioning Code, Data, and Stack

The +d, +u, +c, +s, & +j Options

The linker organizes a program into three sections:

Code	Contains the program's executable code.
Initialized data	Contains those of the program's global and static variables that are assigned an initial value (e.g. `static int var=1`);
Uninitialized data	Contains the program's other global and static variables.

The linker supports options that allow you to position these segments in memory. The `+d 8000` option used in the above command sets the starting address of the program's initialized data to `0x8000`.

The linker's `+u` option sets the starting address of the program's uninitialized data. This option wasn't used in the above command, so the uninitialized data begins at its default address; i.e. immediately above the initialized data.

The linker's `+c` option sets the starting address of the program's code segment. This option wasn't used in the above command, so the code area begins at its default starting address; i.e. location `0`.

The linker's `+s` and `+j` options set the starting address of the program's stack pointer and the size of the stack area. These options weren't used in the above command, so they assume their default values: the stack area begins immediately after the uninitialized data area, the area is 2k bytes long, and the stack pointer initially points at the top of this area.

Naming the Output File: the -o Option

The `-o exmpl` option tells the linker to place the linked program in the file named `exmpl`. If this option wasn't used, the linker would have derived the name of the output file from that of the first object module, by deleting its extension.

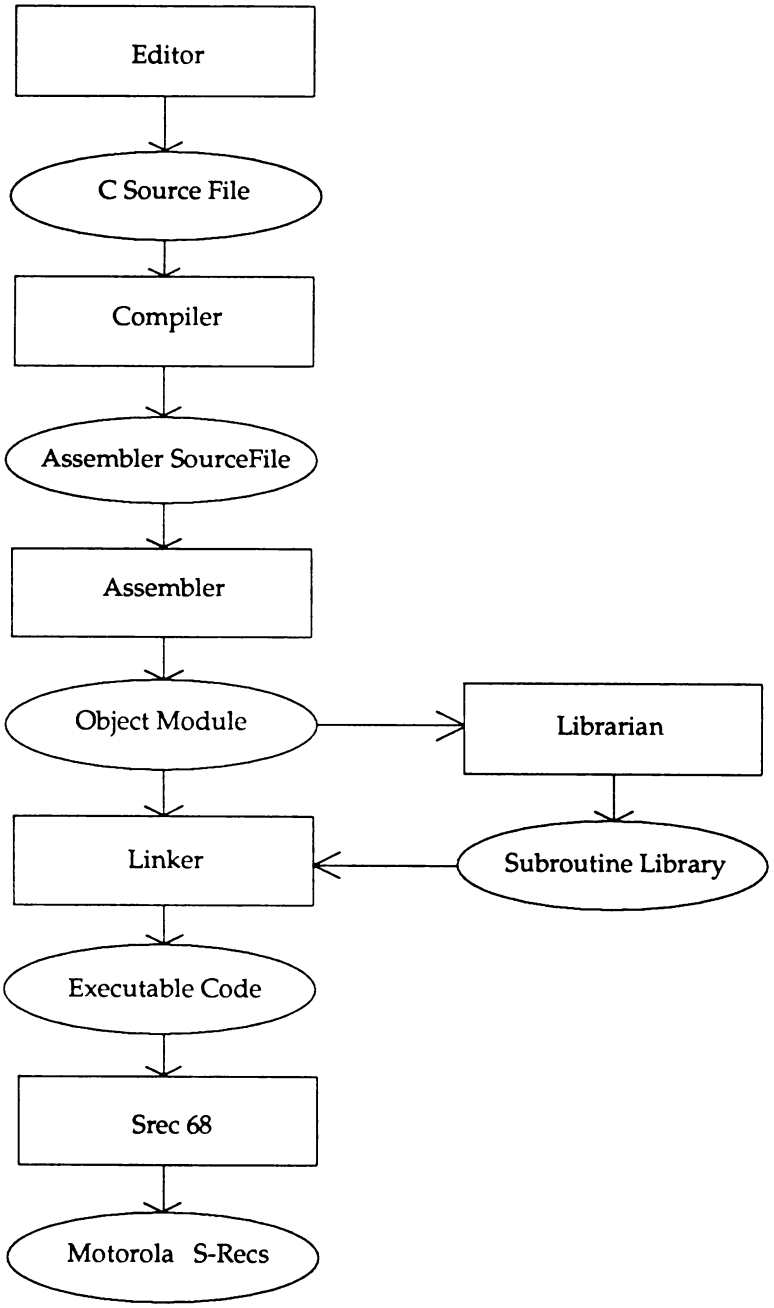
The Input Object Module Files

`exmpl.r` is the name of the file whose object module is to be included in the program.

Libraries and the -l Option

The `-lc` option tells the linker to search the `c.lib` library that's in the directory defined by the CLIB68 environment variable for needed functions.

As you can see, the -l option doesn't completely define the name of a library file; the linker generates the complete name by taking the letters that follow the -l, prepending them with the


Editor
C Source File
Compiler
Assembler SourceFile
Assembler
Object Module
Librarian
Linker
Subroutine Library
Executable Code
Srec 68
Motorola S-Recs

value of the CLIB68 environment variable, and appending the letters .lib. Thus, when CLIB68 has the value `e:\c68\lib\`, the `-lc` option specifies the library whose complete file name is `e:\c68\lib\c.lib`.

During the link step, the linker will search the libraries specified to it for modules containing needed functions; when such a module is found, the linker will include the module in the executable file it is building.

All C programs need to be linked with `c.lib` (or an equivalent, as described below). This library contains the non-floating point functions that are defined in the **Library Functions** chapter. It also contains "internal" functions that are called by compiler-generated code.

If a program performs floating point operations, it must also be linked with the `m.lib` math library (or an equivalent, as described below).

When a program is linked with a math library, that library must be specified before `c.lib`. For example, if `exmpl.c` performed floating point, the following would link it:

```
ln68 +d8000 -o exmpl exmpl.r -lm -lc
```

Putting startup Code First

The startup code for a program sets up the program's environment and then calls the program's `main` function. This code is system dependent, but usually does operations such as the following:

- initialize the program's data segments
- set up the stack
- initialize the heap
- initialize the interrupt vector table.

c.lib contains a generic startup routine. The command that was used above to link the sample program caused the linker to place the startup module in the middle of the program and to create an instruction at the beginning of the program's code segment that jumps to the startup code.

It is often necessary to put a program's startup code at the beginning of the program. For example, the startup code might contain a table of interrupt vectors that is to be located at the beginning of memory. This can be done by linking in the startup module first. The following command demonstrates how a startup module that is in the file `startup.r` can be placed at the beginning of the `exmpl` program's code segment:

```
ln68 +d 8000 -o exmpl startup.r exmpl.r -lc
```

Step 4: Format Conversion

The next step is to convert the memory image generated by the linker into a format that can be loaded into ROM or an ICE. Aztec C68k/ROM has utilities for converting the program generated by the linker into several formats, including Motorola S-records, Intel hex records, Tektronix Ex-

tended Tek Hex Code, and UNIX System 5 COFF format. In the following discussion, we'll generate S-records using **srec68**.

To generate Motorola S-records for the program, enter the following command:

```
srec68 expl
```

When the records generated by this command are fed into a ROM programmer, the resulting ROM code will contain the program's code segment followed by a copy of its initialized data segment.

Note: when the system is started, its RAM contains random values; the Aztec startup routine sets up the RAM-resident initialized data segment from the ROM-resident copy.

These commands generate one or more files, each of which contains S-records for one 2kb, successively-higher addressed section of the program's code and initialized data. The files are **expl.m00** (containing first ROM chip's S-records), **expl.m01** (containing the second ROM chip's S-records), and so on.

srec68 has several additional features. For example, you can explicitly define the size of each ROM chip, using the **-p** option; and you can have it place a program's even-addressed and odd-addressed bytes in separate ROM chips, using the **-e** and **-o** options.

For complete descriptions of srec68 and the other conversion utilities, see the **Utilities** chapter.

Special Features of Aztec C68k/ROM

That concludes our step-by-step, cookbook introduction to Aztec C68k/ROM. In the following paragraphs, we want to introduce several special features of Aztec C68k/ROM.

Memory Models

Aztec C68k/ROM allows you to define, when you compile and assemble a module, the "memory model" that the module will use. A module's memory model affects the module's speed, size, and the amount of data it can access. By default, a module will use the small code, small data memory model, which makes it small and fast, but you can override this using compiler and assembler options.

Here's where to go for more information.

- For a complete description of memory models, see the **Compiler** chapter.
- The compiler options for selecting a module's memory model are **-mc** and **-md**; they are discussed in the "Options" section of the **Compiler** chapter;
- The assembler options for selecting the default memory model are **-c** and **-d**; they are discussed in the "Options" section of the **Assembler** chapter;
- The assembler directives **near** and **far** also define memory models; they are discussed in the "Programmer Information" section of the **Assembler** chapter;
- The creation of libraries is discussed in the **Library Customization** chapter.

Int Size

Aztec C68k/ROM allows you to define, when you compile a module, the size of the **int** data type. By default, an **int** is 32 bits long, but the compiler's **-ps** option can be used to make **ints** 16 bits long.

Libraries

Several libraries are provided with Aztec C68k/ROM, each of which provides different combinations of the following attributes:

1.) Type of Functions:

- functions that perform non-floating point operations (**c**);
- functions that use a 68881 to perform floating point (**m8**);
- functions that use software to perform floating point operations (**m**);

2.) Memory Module Used: either small code/ small data, or large code/ large data;

3.) **int** Size: either 16 bits or 32 bits.

The name of the library is derived from the particular combination of these three attributes that it uses by concatenating the following codes:

- **c, m8,** or **m** depending on the types of functions that are in the library
- **l** If the library's modules use the large code/ large data memory model.
- **16** if the library's modules use 16 bit **int**s

Thus, **c.lib** contains non-floating point functions that use 32 bit **int**s and the small code/ small data memory model. And **m8l16.lib** contains 68881 functions that use 16 bit **int**s and the large code/ large data model.

Register Usage

By default, a program's register usage is as follows:

- Temporary results: data registers **d0-d1**; address registers **a0-a2**.
- Register variables: data registers **d2-d7**; address registers **a3** and **a4**.
- Small model support register: **a5**.
- Frame pointer: **a6**.
- Stack pointer: **a7**.

Using the compiler's **-y** option, you can define the registers used for temporary results, register variables, and the frame pointer.

Using the linker's **+r** option, you can define the register used to support modules that use a small memory model.

The makefiles that are provided with Aztec C68k/ROM generate libraries whose modules use the default registers.

Where To Go From Here

In this chapter, we've just begun to describe the features of Aztec C68k/ROM.

One chapter that you must read is the **Library Customization** chapter, which discusses how to customize the Aztec C68k/ROM library functions for your system.

For more information on the sections of a program, see the **Linker** chapter.

The **srec68** and **hex68** programs support several options that haven't been discussed in this introduction. For a complete description of these programs, see the **Utilities** chapter.

The **Technical Information** chapter contains miscellaneous information on several topics, including the writing of assembly language functions and interrupt handlers.

Refer to the **Compiler**, **Assembler**, and **Linker** chapters, to become familiar with all the options that these programs provide.

Chapter 3 - Compiler

This chapter describes how to use the Aztec C68k/ROM Cross Compiler to produce code for 68000-based target systems. It supports the full C language as defined by the American National Standards Institute (ANSI).

This chapter is organized into the following sections:

- Operating Instructions
- Compiler Options
- Programming Considerations
- Error Messages

Operating Instructions

Use the following command to invoke the Aztec C compiler:

c68 [*options*] *input file*

where [*options*] specifies optional parameters, and *input file* is the name of the file containing the C source program. Options can appear either before or after the name of the C source file.

The compiler reads C source statements from *input file*, translates it to assembly language, and writes the results to an output file.

When the compiler is finished, it activates the Manx assembler, unless option **-a** is used which tells the compiler not to start the assembler. The assembler translates the assembly language source into relocatable object code, writes the result to another file, and deletes the assembly language source file.

The Input File

When you invoke the compiler, the C source input file can be specified as a simple filename or as a complete path specification. The default assumes that the input file is in the current directory. For example, if the file **prog1.c** contains C source and is in the directory **d:\db\source**, you can compile it by using the following command:

```
c68 d:\db\source\prog1.c
```

If the directory containing this file is also the current directory, you could compile the file with the command

```
c68 prog1.c
```

And if the current directory is **db**, on the **d:** drive, you could compile the file with the command

```
c68 source\prog1.c
```

Source Filename Extensions

If the command that starts the compiler does not specify the extension of the file containing the C source, the compiler assumes that the extension is **.c**. For example, the command

```
c68 prog
```

compiles a file named **prog.c** in the current directory.

Although **.c** is the recommended file extension name, it is not mandatory. The specification

```
c68 prog.prg
```

reads the file **prog.prg** from the current directory as the input to the compiler. To specify a program with no file extension at all, you must follow the name with a dot, as in

```
c68 noext.
```

The Output Files

Creating An Object Code File

Normally, when you compile a C program you are interested in the relocatable object code for the program, and not in its assembly language source. Because of this, the compiler by default writes the assembly language source for a C program to an intermediate file and then automatically starts the assembler. The assembler then translates the assembly language source to relocatable object code, writes this code to a file, and erases the intermediate file.

By default, the object code generated by a **c68** started assembly is sent to a file whose name is derived from that of the file containing the C source by changing its extension to **.r**. This file is placed in the directory that contains the C source file. For example, if the compiler is started with the command

```
c68 prog.c
```

the file **prog.r** will be created, containing the relocatable object code for the program.

The name of the file containing the object code created by a compiler-started assembler can also be explicitly specified when the compiler is started, using the compiler's **-o** option. For example, the command

```
c68 -o myobj.rel prog.c
```

compiles and assembles the C source that's in the file **prog.c**, writing the object code to the file **myobj.rel**.

When the compiler is going to automatically start the assembler, it by default writes the assembly language source to a temporary file named *ctmpxxx.xxx*, where the x's are replaced by digits in such a way that the name becomes unique. This temporary file is placed in the directory specified by the environment variable CCTEMP. If this variable doesn't exist, the file is placed in the current directory.

When CCTEMP exists, the complete name of the temporary file is generated by simply prefixing its value to the *ctmpxxx.xxx* name. For example, if CCTEMP has the value

```
e:\temp
```

then temporary files are placed in the **e:\temp** directory.

An environment variable is created using the **set** command. For example, the following commands create CCTEMP and assigns it the value **e:\temp**:

```
set CCTEMP = e:\temp
```

Creating Just An Assembly Language File

There are some programs for which you don't want the compiler to automatically start the assembler. For example, you may want to modify the assembly language generated by the compiler for a particular program. In such cases, you can use the compiler's **-a** option to prevent the compiler from starting the assembler.

When you compile a program using the **-a** option, you can tell the compiler the name and location of the file to which it should write the assembly language source, using the **-o** option.

If you don't use the **-o** option but do use the **-a** option, the compiler will send the assembly language source to a file whose name is derived from that of the C source file by changing the extension to **.a** and place this file in the same directory as the one that contains the C source file.

For example, the command

```
c68 -a prog.c
```

compiles, without assembling, the C source that is in **prog.c**, sending the assembly language source to **prog.a**.

As another example, the command

```
c68 -a -o temp.asm prog.c
```

compiles, without assembling, the C source that is in **prog.c**, sending the assembly language source to the file **temp.asm**.

When the **-a** option is used, the option **-t** causes the compiler to include the C source statements as comments in the assembly language source.

#include Files

Searching For #include Files

You can make the compiler search for **#include** files in a sequence of directories, thus allowing source files and **#include** files to be contained in different directories.

Directories can be specified with the **-i** compiler option, and with the INCL68 environment variable. The compiler itself also selects a few areas to search. The maximum number of searched areas is eight.

If the file name in the **#include** statement specifies a directory, just that directory is searched.

The -i option.

A **-i** option defines a single directory to be searched. The area descriptor follows the **-i**, with no intervening blanks.

For example, the following **-i** option tells the compiler to search the **d:\az68\include** directory:

```
-id:\az68\include
```

The INCL68 Environment Variable

The INCL68 environment variable also defines directories to be searched for **#include** files. The string associated with this variable consists of the names of the directories to be searched, with each pair separated by a semicolon. For example, the following command sets INCL68 so that the compiler will search for include files in directories **e:\include** and **d:\az68\include**:

```
set INCL68=e:\include;d:\az68\include
```

The Search Order For Include Files

Directories are searched in the following order:

- If the **#include** statement delimited the file name with the double quote character, ", the current directory on the default drive is searched. If delimited by angle brackets, < and >, this area is not automatically searched.
- The directories defined in **-i** options are searched, in the order listed on the command line.
- The directories defined in the INCL68 environment variable are searched, in the order listed.

Precompiled header Files

To shorten compilation time, the compiler supports precompiled **#include** files.

To use this feature, you first compile frequently-used header files, specifying the **-hi** option; this causes the compiler to write its symbol table, which contains information about the contents of the header files, to a disk file. Then, when you compile a module that **#includes** some of these header files, you specify the **-hi** option; this causes the compiler to load into its symbol table the pre-compiled symbol table information about the header files. When the compiler encounters a **#include** statement of a header file for which it has already loaded pre-compiled symbol table information, it ignores the **include** statement. This ignoring occurs even if the **#include** file was nested within another **#include** file in the C source from which the pre-compiled symbol table was generated.

The compiler does much less work when it loads pre-compiled information into its symbol table than when it generates the same information from C source, and hence using pre-compiled **#include** files can considerably shorten the time required to compile a module.

The **-ho** option tells the compiler to write its symbol table to a file. The name of the file follows the **-ho**; spaces can optionally separate the **-ho** and the filename. For example, you might create a file named **x.c** that consists just of **#include** statements for all the header files that you want pre-compiled. You could then generate a file named **include.pre** that contains the symbol table information for these header files by entering the following command:

```
c68 -ho include.pre x.c
```

The **-hi** option tells the compiler to read pre-compiled symbol table information from a file. The name of the file follows the **+i**; spaces can optionally separate the **-hi** and the filename. For example, to compile the file **prog.c** that accesses the header files that were defined in **x.c**, and to have the compiler preload the symbol table information for these files from **include.pre**, enter the following command:

```
c68 -hi include.pre prog.c
```

When reading a pre-compiled header file during compilation of a file, you must take care that the compiler options used for generating the pre-compiled header file are compatible with those used for the current compilation.

Only one pre-compiled header file can be read during compilation of a file.

Memory Models

The memory model used by a program determines how the program's executable code makes references to code and data. This in turn indirectly determines the amount of code and data that the program can have, the size of the executable code, and the program's execution speed.

Before getting into the details of memory models, we want to describe the sections into which a program is organized. The sections of a program are these:

- CODE, containing the program's executable code;
- DATA, containing its global and static data;
- STACK, containing its automatic variables, control information, and temporary variables;
- HEAP, an area from which buffers are dynamically allocated.

There are two attributes to a program's memory model: one attribute specifies whether the program uses the large data or the small data memory model; the other attribute specifies whether the program uses the large code or small code memory model.

Large Data Versus Small Data

The fundamental difference between a large data and a small data program concerns the way that instructions access data segment data: a large data program accesses the data using position-dependent instructions; a small data program accesses the data using position-independent instructions. An instruction makes position-dependent reference to data in the data segment by specifying the absolute address of the data; it makes a position-independent reference to data in the data segment by specifying the location as an offset from a reserved address register. Other differences in large data and small data programs result from this fundamental difference; these other differences are:

There is no limit to the amount of global and static data that a large data program can have. A small data program, on the other hand, can have at most 64k bytes of global and static data.

For a small data program, an address register (which we call the SMALL MODEL SUPPORT REGISTER) must be reserved to point into the middle of the data segment. For a large data program, an instruction that wants to access data in the data segment contains the absolute address of the data, and hence doesn't need this address register.

A code segment is larger when its program uses large data than when it uses small data, because a reference to data in a data segment occupies a 32-bit field in a large data instruction, and occupies a 16-bit field in a small data instruction.

A program is slower when it uses large data than when it uses small data, because it takes more time for an instruction to access data when it specifies the absolute address of the data than when it specifies the data's offset offset from an address register.

Large Code Versus Small Code

The fundamental difference between a large code and a small code program concerns the way that instructions in the program refer to locations that are located in the code segment: for a large code program the reference is made using position-dependent instructions; for a small code program, the reference is made using position-independent instructions. An instruction makes position-dependent reference to a code segment location by specifying the absolute address of the location; it makes a position-independent reference to a code segment location by specifying the location as an offset from the current program counter. Other differences in large data and small data programs result from this fundamental difference; these other differences are:

The size of a code segment is unlimited for both large code and small code programs. An instruction in a large code program can directly call or jump to the location, regardless of its location in the code segment.

An instruction in a small code program can only directly call or jump to locations that are within 32k bytes of the instruction. To allow instructions in small code programs to transfer control to any location, regardless of its location in the code segment, a "jump table", which is located in the program's data segment, is used. If a location to which an instruction wants to transfer control is more than 32k bytes from the instruction, the transfer is made indirectly, via the jump table: the instruction calls or jumps to an entry in the jump table, which in turn jumps to the desired location.

A jump instruction in a jump table entry refers to a code segment location using an absolute, 32-bit address, and hence can directly access any location in the program's code segment.

When a small code program is linked, the linker automatically builds the jump table: if the location to which an instruction wants to transfer control is outside the instruction's range, the linker creates a jump table entry that jumps to the location and transforms the pc-relative instruction into a position-independent call or jump to the jump table entry.

A code segment can contain data as well as executable code. An instruction in a large code program can access data located anywhere in the code segment, because it accesses code segment data using position-dependent instructions, in which the location is referred to using a 32-bit, absolute address. An instruction in a small code program can only access code segment data that is located within 32k bytes of the instruction.

For a small code program to access the jump table, an address register needs to be reserved and set up to point into the middle of the program's data segment; if the program also uses small data, the same address register (that is, the SMALL MODEL SUPPORT REGISTER) is used for both jump table accesses and normal accesses of data segment data. For a large code program, this address register is not needed for the referencing of locations in the code segment.

A code segment is larger when its program uses large code than when it uses small code, because instructions that reference code segment locations by specifying an absolute address use a 32-bit field to define the location, whereas instructions that reference data by specifying a pc-relative address or an offset from an index register use a 16-bit field to define the location.

A program is usually slower when it uses large code than when it uses small code, because it takes more time for an instruction to reference a code segment location when it specifies the absolute address of the data than when it specifies the location in a pc-relative form.

A large small code program that has lots of indirect transfers of control via the jump table may not differ much in execution time from a large code version of the same program, since the small code indirect transfer via the jump table will take more time than the large code direct transfer.

Selecting A Module's Memory Model

You define the memory model to be used by a module when you compile the module, by specifying or not specifying the following options:

-mc Module uses large code. If this option isn't specified, the module will use small code.

-md Module uses large data. If this option isn't specified, the module will use small data.

For example, the following commands compile **prog.c** to use different memory models:

c68 prog	small code, small data
c68 -mc prog	large code, small data
c68 -md prog	small code, large data
c68 -mc -md	large code, large data

Multi-Module Programs

The modules that you link together to form an executable program can use different memory models, with the following caveat.

When large data and small data modules are linked together, the linker will create an arbitrarily large data segment, without attempting to sort the data into those that are accessed by large data modules and those that are accessed by small data modules. When the program is running, an address register that you specify at link time will point into the middle of this data segment. This register is used by the small data modules to access data.

Here's the caveat: data that the small data modules attempt to access must be within 32k bytes of the location pointed at by this address register. The linker will detect data accesses by small data modules for which this condition isn't satisfied, and issue a message. If you get this message, try reordering the order in which the linker encounters them; if that doesn't solve the problem, you'll have to recompile the small data modules, making them use large data.

Compiler Options

Option Format

Most of the compiler options are set up as toggles, which means that they can be either on or off. Most options default to off. The defaults can be changed by creating an environment variable, CCOPTS. Options specified directly with the compile command will override options specified through the CCOPTS environment variable.

With a few exceptions, options are grouped around a common function. The first letter of an option identifies the group. The group letters are:

- **a** Assembly language output control
- **b** Debugging control
- **c** Target chip (processor) control
- **f** Floating point control
- **h** Precompiled header file control
- **m** Memory model control
- **p** Parser control
- **q** Prototype generation, etc.
- **r** Register control
- **s** Optimization control
- **w** Warning control

After the group letter, one or more individual options may be specified. If an individual option letter occurs and is NOT preceded by a **0** (zero), the associated option is turned on. Multiple individual options can be specified.

To turn an option off, the character **0** (zero) must appear after the group letter and before the options to be turned off. For example, **-p0t** turns trigraphs off and **-pt** turns trigraphs on. The **0** remains in effect for the remainder of the options specified after the group letter, or until a **1** is encountered. Thus **-p0td** turns off both the **-pt** and **-pd** options, and **-p0td1b** turns off the **-pt** and **-pd** options and turns on the **-pb** option.

Combinations of options can be used to produce very specific results. To enable full ANSI syntax checking with the singular exception of trigraphs, for example, you would use the option **-pa0t**. The **a** option of the **p** group specifies full ANSI which includes trigraphs. The **0t** option turns trigraphs off. Since options are scanned left to right the combination **-pa0t** would produce the desired result. **-p0ta** would not produce the intended result; since the **a** option is scanned after the **0t** option, the **0t** option would be cancelled.

CCOPTS Environment Variable

You can specify options to the compiler by using the environment variable CCOPTS. The compiler looks at the CCOPTS variable first for its options, then looks at which options were passed

directly to it when it was called. Options passed directly to the compiler override the CCOPTS environment variable. For example, if the CCOPTS environment variable was set to **-ws** and you specified **-w0s** as a direct option to the compiler, then the CCOPTS **-ws** option would be overridden.

WARNING: Options that require additional arguments must not be specified using the CCOPT environment variable. These include the **-o, -d, -i,** and **-h** options.

The following list describes all the available options. Unless specified, the options default to off. Options whose descriptions end in an * are described in more detail in the following section.

Option Summary

-a Do not start the assembler after compilation is done. Just generate an assembler source file. *

-at Same as -a, but also imbed C source statements into the assembly code.

-bd Generate stack depth-checking code on function entry.*

-bs Generate source debugging information.

-c2 Generate 68020 code.*

-d ***symbol*** **[=*value*]** Define a symbol for the preprocessor.*

-fm Generate code for Manx IEEE floating point. **Defaults to on.***

-f8 Generate code for 68881 Floating Point format. *

-hi *file* Read precompiled symbol table from *file*. For more information on **-hi**, see the section "Precompiled **#include** Files" discussed earlier in this chapter.

-ho *file* Write symbol table to *file*. For more information on **-ho**, see the section "Precompiled **#include** Files" discussed earlier in this chapter.

-i *dir* Search directory, *dir* for **#include** files. For more information, see the section "Precompiled **#include** Files" discussed earlier in this chapter.

-k Enable support for the K&R(Unix V7) specific features of C, and disable support for the ANSI specific features. See also **-pa** option.*

-mb Generate the public **.begin** statement. **Defaults to on.** *

-mc Generate code that uses the large code memory model. For information see the discussion of memory models in the section "Operating Instructions".

-md	Generate code that uses the large data memory model.
-me	Align strings on word boundaries. This forces all strings to start on an even address.
-mm	Put initialized data in the code segment instead of the data segment.*
-mp	Align structure elements on word boundary. **Defaults to on.**
-mr	Add more temporary registers to handle a complex expression.*
-ms	Put string constants in data segment.
-o *file*	Write output to *file*. For details, see the section "Operating Instructions".
-pa	Enable support for the ANSI specific features of C, and disable support for the K&R specific features. Turns off **c,d,e,k,o,u** suboptions for **-p.** See also **-k** option.*
-pb	Make bitfields unsigned by default.*
-pc	Allow extra characters after a **#endif** or **#else. Defaults to on.**
-pd	Allow use of direct functions. For details, see the section "Programming Considerations". **Defaults to on.***
-pe	Cause **enums** to occupy only the amount of space needed. **Defaults to on.***
-pk	Enable the use of functions that use Pascal calling conventions. For details, see the section "Programming Considerations". **Defaults to on.**
-pl	Make ints 32 bits. **Defaults to on.** See also **-ps** option.*
-po	Use old K&R (3.6) preprocessor.
-pp	Make characters unsigned by default instead of signed.
-ps	Make ints 16 bits long. See also **-pl** option.
-pt	Enable trigraph support.*
-pu	Use unsigned preserving rules instead of value preserving.*
-qa	Cause generated prototypes to use __PARMS(()) syntax.*
-qp	Generate prototypes for all non-static functions defined within the current file.

-qq	Be quiet.*
-qs	Generate prototypes for all static functions defined within the current file.
-qv	Be verbose.*
-sa	Enable two pass assembly for branch squeezing and other optimizations.
-sb	Generate in-line code for the functions **strcpy()**, **strcmp()**, and **strlen()**.
-sf	Generate an optimized **for(;;)** loop that places the test at the bottom of the loop.*
-sm	Define the __C_MACROS__ macro to replace some functions with in-line macro expansions defined in the header files.
-sn	If no local variables are created on the stack for a function, do not generate the **link** and **unlk** instructions.*
-so	Perform full optimization. **-so** is equivalent to **-sabfmnprs**.
-sp	Delay the popping of arguments until necessary.*
-sr	Automatically allocate registers based on weighted usage counts.*
-ss	Find duplicate string literals and replace them with a pointer to the first occurrence.*
-su	Automatically allocate registers based on weighted usage counts, but allocate to user specified variables first.*
-wa	Complain on arguments which do not match the prototype specification.*
-wd	Generate warnings for old style K&R function definitions.*
-we	Treat warnings as errors.
-wf	Do not generate warnings for static functions that are declared but not defined. *
-wl	Perform maximum type checking. Equivalent to **-waruf.***
-wn	Do not generate warnings on direct pointer to pointer conversions.*
-wo	Cause pointer/integer conflicts to generate warnings rather than errors.*
-wp	Generate a warning if a function is called when a prototype for a function is not visible.*

-wq	Print warnings to the file **aztecC.err** instead of to the screen.
-wr	Warn if function return type does not match declared type.*
-ws	Ignore all warnings.
-wu	Warn about unused local variables.*
-ww	Print all error messages without pausing.
-yda*x*	Use address register **a***x* as the data pointer. Default: **a5.**
-yfa*x*	Use address register **a***x* as the frame pointer. Default: **a6.***
-yr*reglist*	For register variables, use the registers defined by *reglist*. Default: **d2-d7/a3-a4.***
-ys*reglist*	On function entry, always save the registers defined by *reglist*. Default: none.*
-yt*reglist*	For temporary results, use the registers defined by *reglist*. Default: **d0-d1/a0-a2.***
-yu*x*	Define method for translating C symbols to assembler:*

Option Descriptions

Option -a

In some programs, you may not want the compiler to start the assembler automatically. For example, you may want to modify the assembly language generated by the compiler for a particular program. In such cases, use compiler option **-a**, which prevents the compiler from starting the assembler.

When you specify option **-a**, by default the compiler sends the assembly language source to a file whose name is derived from that of the C source file, by changing the extension to **.asm**. This file is placed in the same directory as the one that contains the C source file. For example, the command

```
c68 -a prog.c
```

compiles, without assembling, the C source that is in **prog.c**, sending the assembly language source to **prog.asm**.

When using option **-a**, the **-o** option specifies the name of the file to which the assembly language source is sent. For example, the command

```
c68 -a -o e:temp.asm prog.c
```

compiles, without assembling, the C source in **prog.c**, sending the assembly language source to the file **temp.asm** on the **e:** drive. When option **-a** is used, sub-option **t** causes the compiler to include the C source statements as comments in the assembly language source. In other words, you would use **-at** to request the compiler to insert the original C source as comment code when it generates the assembly file.

In other words, if you are interested in the assembler source you would specify option **-at** when you start the compiler. This causes the compiler to send the assembly language source to a file whose name is derived from the file containing the C source, and whose extension is set to **.asm**. The C source statements are included as comments in the assembly language source. For example, the command

```
c68 -at prog.c
```

compiles **prog.c**, creating the file **prog.asm** with the C source inserted as comments.

Option -bd

The **-bd** option causes the compiler to generate code that performs stack depth-checking on function entry. To do this the compiler generates a call to the assembly language function **_stkchk()**. If **_stkchk()** detects that the stack has grown too large, it calls the C-language routine **_stkover()**.

You will need to modify **_stkover()**, since the supplied version simply returns. For example, **_stkover()** could print an error message and then exit.

Since compiling with **-bd** causes the code to be bigger and execute slower, the final version of your program should be compiled without this option.

Option -c2

The **-c2** option causes the compiler to generate 68020 instructions that take advantage of the 68020 and 68030 chips. These instructions will not run on a 68000 or 68010.

Option -d

Option **-d** defines a symbol in the same way as the preprocessor directive, **#define**. Its usage is as follows:

c68 -d*macro*[=*text*] *filename*

For example,

```
c68 -dMAXLEN=1000 prog.c
```

is equivalent to inserting the following line at the beginning of the program:

```
#define  MAXLEN  1000
```

The separating space following the **-d** is optional. The following formats are equivalent:

```
-d MAXLEN=1000
-dMAXLEN=1000
```

Since option **-d** causes a symbol to be defined for the preprocessor, it can be used in conjunction with the preprocessor directive, **#ifdef**, to selectively include code in a compilation. A common example is code such as the following:

```
#ifdef  DEBUG
  printf("value: %d\n", i);
#endif
```

This debugging code would be included in the compiled source by the following command:

```
c68 -dDEBUG program.c
```

When no substitution text is specified, the symbol is defined as the numerical value one.

This capability is useful when small pieces of code must be altered for different operating environments. Rather than maintaining two copies of such a program, this compile time switch can be used to generate the code needed for a specific environment.

Option -fm

The **-fm** option causes the compiler to generate code that performs floating point operations by calling internal routines that are in **m.lib**. Floating point numbers are represented in 68881 format.

NOTES

- When a program's modules are compiled with **-fm**, the program must be linked with the appropriate version of **m.lib**.
- All of a program's modules must use the same **-f** option.

Option -f8

The **-f8** option causes the compiler to generate code that uses the 68881 math co-processor. The generated code is in-line when possible; otherwise it is in internal routines within versions of **m8.lib** and the compiler generates an in-line call to the internal routines.

NOTES

- When a program's modules are compiled with **-fm8**, the program must be linked with the appropriate version of **m8.lib**.
- All of a program's modules must use the same **-f** option.

Option -k

The **-k** option causes the compiler to adhere to K&R (UNIX version 7) C syntax, rather than ANSI. This option is useful in compiling code written in Aztec C version 3.6 or some other K&R conforming compiler. The **-k** option is equivalent to compiling with the options **-pou0a -wo.**

Option -mb

The standard startup code for a program is in the versions of **c.lib** in the **rom68** module. **.begin** is the name of the entry point in this startup code. By default to force the linker to include in a program startup code that is in the **rom68** module, the compiler generates a reference to **.begin**.

The **-mb** option controls whether or not the compiler generates a reference to **.begin**:

- **-mb** causes the compiler to generate a reference to **.begin**, and
- **-m0b** prevents the compiler from generating a reference to **.begin**.

NOTES

The libraries have been compiled with **-m0b**.

Option -mm

The **-mm** option causes the compiler to put initialized data in the program's code segment instead of the initialized data segment.

NOTES

This option does not have any effect on the placement of unititialized variables.

Option -mr

This option should only be used when the compiler issues an error message that an expression is too complex. It increases the number of scratch or temporary registers that are available. On the down side, it reduces the number of register variables that can be allocated to registers.

For ROM based applications that explicitly specify register usage, the **-yt** option should be used in conjunction with the **-yr** option instead of the **-mr** option to increase the number of temporary registers available.

Option -pa

This switch is used to turn on all the ANSI checking and turn off any special extensions. A program which compiles with the **-pa** flag without errors or warning should compile cleanly with any other ANSI standard C compiler.

Option -pb

This option causes bit fields to be treated as unsigned. The version 5 default is signed. For example

```
int pFlags :3;
```

by default defines a bit field whose value ranges from **-4** to **+3**. If the **-pb** option is specified, the range is from **0** to **+7**.

Option -pd

The **-pd** option enables support for direct functions. For more information, see the discussion of direct functions that appears later in this chapter.

Option -pe

The **-pe** option places **enums** in the smallest space that will contain them. For example, the definition:

```
enum colors {blue, red, yellow, white};
```

would use a single byte to represent **enums** of this type if the **-pe** option was used. The definition:

```
enum countries {USA, England, France=500, Germany}
```

would use a two byte word to represent **enums** of this type when using the **-pe** option. This option defaults to on.

Options -pl and -ps long

The **-pl** and **-ps** options determine the size of an **int**:

Option	Int size
-pl	32 bits
-ps	16 bits

By default, **int**s are 32 bits long. A program's modules must all use the same size for **int**s. This includes library modules; for more information on the libraries that are provided with Aztec C68k/ROM, see the **Tutorial** chapter.

Option -pt

Trigraphs are used to support foreign keyboards and printers. Certain characters are represented as two question marks followed by another character which indicates the true character intended. For example, **??=** is equivalent to **#** and **??(** is equivalent to **[**. In most instances, this simply slows down the compiler, but it must be supported for ANSI compliance.

The **-pt** option determines whether or not trigraphs are supported: **-pt** enables trigraphs and **-p0t** disables trigraphs. By default, trigraphs are disabled.

Option -pu

Aztec C68k/ROM can evaluate expressions using either value-preserving or unsigned-preserving rules:

- With value-preserving rules, all data types are assigned a rank. When two operands are used in an expression, the operand having lesser rank is promoted to the type of the other operand. The ranking of data types, ordered from lowest to highest is as follows: **char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double**. Value-preserving rules are ANSI compatible.

- With unsigned-preserving rules, all signed data types are assigned a rank. When two operands are used in an expression, the expression is first evaluated as if both operands are signed, after promoting the lesser-ranked operand to the type of the other operand. Then, if either operand is unsigned, the type of the expression is set to unsigned. Unsigned preserving rules are K & R compatible.

Value-preserving rules, which the compiler uses by default can be explicitly enabled using the **-p0u** option or the **-pa** option. Unsigned-preserving rules can be enabled using the **-pu** option or the **-k** option.

As an example of the difference in these rules, consider the addition of an **unsigned char** to a **signed int**: for value-preserving rules, the type of the sum is **signed int**, while for unsigned-preserving rules, it is **unsigned int**.

Option -qa

The **-qa** argument controls how the prototypes are generated. If **-qa** is specified, then a prototype is generated as:

int *func* **_PARMS((***int x, int y***));**

instead of the default:

int *func***(***int x, int y***);**

The first form is used for maximum portability to pre-ANSI compilers. The following sequence is usually placed at the beginning of a header file containing these style prototypes:

```
#if __STDC__
#define _PARMS(x) x
#else
#define _PARMS(x) ()
#endif
```

Then, if __STDC__ is defined, the macro will change the above prototype to:

int *func* **(***int x, int y***);**

Otherwise if __STDC__ is not defined, the prototype becomes:

int *func* **();**

which is the standard declaration of an external function defining the type returned by the function.

Option -qq

The **-qq** option prevents the compiler from generating the signon message that is normally displayed when the compiler starts and the count of the total number of errors that is normally displayed before the compiler stops.

Option -qv

The **-qv** option causes the compiler to be verbose; during compilation it displays information about what it is doing and before stopping it displays information about memory usage.

Option -sf

Versions prior to 5.0 of the Aztec C compiler always generated **for(;;)** loops with the tests and increment at the bottom of the loop since this is smaller and faster than having the test and increment at the top. However, if you had a statement of the type:

```
for (i=0;i<n;i++)
  do_something;
```

then **sdb** would treat the increment and the test as part of the statement since there is no "}". As a result, if you tried to step through the loop, the first **s** or **t** command would cause the entire loop to be executed.

Now, the compiler defaults to using the previous style **for(;;)** loop generation where the test and increment are at the top of the loop and are thus associated with the **for(;;)** statement, which is better for **sdb**. The **-sf** option, however, generates faster smaller code.

Option -sn

A function's arguments and auto variables are stored in an area that is reserved for the function, called the FRAME. Normally, a function accesses the information that is stored in its frame using a register that is reserved for this purpose, called the FRAME POINTER REGISTER. However, since there are usually several functions active at any time, and there is only one frame pointer register, the frame pointer register must usually be preserved on entry to a function and restored on exit.

The **-sn** option causes the compiler to generate code for a function that accesses the function's frame using the stack pointer register instead of the frame pointer register. This makes the program run faster, since the frame pointer register need not be presented.

Option -sp

The **-sp** option delays stack cleanup until it is absolutely necessary. This speeds up the program and makes it smaller, by combining several cleanup operations into one.

Option -sr

Option **-sr** causes the compiler to automatically assign local variables to machine registers, even if they were not declared with the **register** keyword. If there are more local variables than there are available registers, the compiler will pick the most heavily used variables to place in registers. This option can produce significant savings in code size and execution speed. This option works in conjunction with the **-sn** option since no local storage is required if all variables end up in registers.

Option -ss

This option finds duplicate string literals and replaces them with a pointer to the first occurrence. This also works for common tail subsets of strings. For example, if the two strings **highway** and **way** are used in a function, the code generated will use a pointer to the fifth letter of **highway** for the **way** string. If the string **high** also occurred, it would be stored separately, since it would not match.

Option -su

This is the same as the **-sr** option, except that **-sr** ignores any register statements when picking registers, while **-su** uses these first and then allocates any remaining registers.

Option -wa

Normally, if you call a function in the presence of a prototype, and the type of a function parameter does not match the type specified in the prototype, the type is coerced as if by assignment. Thus if you pass an **int** to a function expecting a **long**, the **int** is quietly cast to a **long**. This option causes the compiler to generate a warning if a quiet cast is generated. This is useful, since it allows you to change the code to an explicit cast which is portable to pre-ANSI compilers.

Option -wd

This option generates a warning if a function is defined using the old pre-ANSI style definition of the form:

```
int function (a, b)
int a, b;
{}
```

instead of:

```
int function (int a, int b)
    {}
```

This a useful tool for converting your programs to the ANSI standard.

Option -wf

By default, the compiler generates a warning if a static function is declared but not defined. The **-wf** option suppresses this warning. This option is used when compiling the output of some C++ translators, which generate such code.

Option -wl

The **-wl** option is a short-hand way of specifying the **-wa**, **-wr**, and **-wu** options. It is used to force the compiler to do maximum type checking.

Option -wn

The **-wn** option suppresses warning messages for direct pointer to pointer conversions which do not contain an explicit cast, such as the following code fragment:

```
int *iptr;
char *cptr;
cptr = iptr;
```

Because the ANSI Standard defines that such direct conversions are illegal, the **-pa** (full ANSI) option will always generate an error for the above code. To eliminate the error when **-pa** is specified, the above could be changed to:

```
int *iptr;
char *cptr;
cptr = (char*)iptr;
```

Option -wo

Under ANSI C, pointer-integer conversions are illegal, and are usually a problem when 16 bit **int**s are used. The **-wo** option changes pointer-integer conversion errors into warnings, and has been included for compatibility with version 3.6.

Option -wp

The **-wp** option causes the compiler to issue a warning message if a function is called which does not have a function prototype. This option is useful for determining if a header file is not being included. For example, if you use **strlen()** without including **string.h**, then **-wp** will generate a warning since the prototype for **strlen()** is contained in the **string.h** header file.

Option -wr

If the -wr option is specified, the compiler will generate a warning message for any functions declared as returning a value but which do not actually return an explicit value. This includes functions which have an implied int return value. Thus, the function:

```
foo()
{
   printf("hello\n");
}
```

would generate a warning since it should have been declared:

```
void foo()
```

since there is no value returned. The warning would be on the }, which is an implicit return statement.

Option -wu

This option directs the compiler to check for unused local variables within functions and generates a warning for each that is not used. For example, the function:

```
void foo()
{
   int j;
   printf("hello\n");
}
```

would generate a warning that `j` is not used in the function.

Option -ww

Normally the compiler will pause after encountering five errors and ask if compilation should continue. The -ww option indicates that the compiler should not pause and should continue to the end of compilation.

Option -yf

During execution of a function, a "frame" of information about the function is on the stack. An address register points to the "frame" of the currently-active function, and is used by compiler-generated code to access information in this function's frame.

You can define, using the compiler's **-yf** option, the address register that will contain the frame pointer. The name of this address register follows the **-yf**, with optional intervening spaces. For example, the following option tells the compiler to use address register **a5** as the frame pointer:

```
-yfa5
```

If this option isn't specified, address register **a6** is used as the frame pointer.

Option -yr

The **-yr** option defines the registers that can be used for a C function's register variables. These registers are specified by the register list that immediately follows the **-yr**. This list is in standard 68000 assembly language format.

For example, the following option defines registers **d4-d7/a3-a5** as register variables:

```
-yrd4-d7/a3-a5
```

If you don't specify the **-yr** option, the compiler uses registers **d2-d7/a3-a4** for register variables.

Option -ys

On entry to a function, the contents of the registers that hold the function's register variables are pushed on the stack. Normally, just those registers that contain the function's register variables are saved; for example, if **d4-d7/a3-a4** are available for use as register variables but the function only declares one register variable, then just one register is saved on entry to the function.

The **-ys** option tells the compiler to generate code that will automatically save specified registers, whether or not they are used for the function's register variables. These registers are specified in a register list that follows the **-ys**.

For example, the following option tells the compiler to generate code that will automatically save **a5** on entry to a function:

```
-ysa5
```

If this option isn't specified, no extra registers will be saved on entry to a function.

Option -yt

During the execution of compiler-generated code, registers are used to hold temporary values. This option defines those registers, in a register list that immediately follows the **-yt**.

For example, if **d0-d2/a0** are available for temporaries, the following **-yt** option would be used:

```
-ytd0-d2/a0
```

If this option isn't specified, registers **d0-d1/a0-a2** will be used for temporaries.

Option -yu

When the compiler translates the name of a function or global variable into assembler, it does so by pre-pending an underscore to the C name, post-pending an underscore, or by using the C name as is. The **-yu*x*** option defines which of these choices the compiler should use, as follows:

x:	**Translate C to Assembler by ...**
p	prepending underscore
a	appending underscore
n	no underscore

If this option isn't used, the compiler will prepend an underscore.

The assembly language routines that are in the library have been written using labels that prepend an underscore. If the **-ya** or **-yn** option is used, the labels in these routines must be modified accordingly. The libraries themselves must be built using the same underscore option for all modules.

Programming Considerations

The previous sections of this chapter discussed operational features of the compiler, that is, information that an operator would use to compile a program. In this section, information is presented which is of use to those who are actually writing the programs.

Supported Language Features

Aztec C68k/ROM is entirely compatible with the ANSI standard for C.

Aztec C68k/ROM also has a number of compiler options and library functions that provide a fair level of compatibility with UNIX v7, UNIX System 3, UNIX System V, and Xenix.

Data Formats

The following paragraphs describe the data formats used by Aztec C68k/ROM.

char

Variables of type **char** are one byte long, and can be signed or unsigned. The default SIGNEDNESS for **char** is signed, but can be made unsigned using the **-pp** option.

When a signed **char** variable is used in an expression, it is converted to a 16-bit integer by propagating the most significant bit. Thus, a **char** variable whose value is between 128 and 255 will appear to be a negative number if used in an expression.

When an unsigned **char** variable is used in an expression, it is converted to a 16-bit integer in the range 0 to 255.

A character in a **char** is in ASCII format.

Aztec C68k/ROM supports character constants that contain more than one character. For example, the character constant **1234** is equivalent to the hex value **0x31323334**.

NOTE:
A character constant that contains multiple characters is not a string!

pointer

pointer variables are four bytes long.

short

Variables of type **short** are two bytes long. They can be signed or unsigned, and by default are signed.

A negative value is stored in two's complement format. A **short** is stored in memory with its least significant byte at the highest numbered address. A **-2** stored at location **100** would look like:

location	contents in hex
100	FF
101	FE

long

Variables of type **long** occupy four bytes, and can be signed or unsigned.

Negative values are stored in two's complement format. Longs are stored sequentially with the most significant byte stored at the lowest memory address and the least significant byte at the highest memory address.

int

int variables can be either 16 or 32 bits long, as determined by the **-ps** and **-pl** options. By default, **int**s are 32 bits long.

float, double, and long double

float, double, and **long double** numbers are represented using 68881 format occupying respectively 4, 8, and 12 bytes of storage .

Structures

By default, all structure elements are aligned on **word** boundaries. This can result in the presence of padding bytes, also known as HOLES, within a structure. For example, consider the following structure:

```
struct a_struct  {
   char a_char;
   int an_int;
};
```

In this structure, a padding byte will exist between the **a_char** and **an_int** elements to insure that **an_int** falls on an even address boundary. By default, structures are always made to contain

an even number of bytes; this is done, when necessary, by adding padding bytes at the end of the structure. For structures that contain just characters, this padding at the end of the structure can be disabled using the **-m0p** option.

Because of the existence of holes within structures, the **sizeof** operator will not always return what you expect. In the above example,

```
sizeof (a_struct)
```

would return (assuming 32 bit ints) a **6**, not a **5**. This may cause problems in porting code from environments such as the 8086 and 8-bit micros.

Bitfields

Bitfields are fully supported by Aztec C. Bitfields are numbered as per normal 68000 convention. For example:

```
struct {
   int bf1 :1;
   int bf2 :1;
   int bf3 :4;
} bf;
```

Here **bf1** will correspond to bit **0** in the first word in the structure **bf, bf2** will correspond to bit **1**, and **bf3** will correspond to bits **2-5**. As many bit fields as possible are put into a single word location. If a bitfield entry has a length and position such that it would straddle two words, it will be moved entirely into the second word, and the remaining bits in the first word are considered to be undefined.

The default signedness for bitfields can be either signed or unsigned, as determined by the **-pe** option. If this option is not used, bitfields are signed by default.

Enum Constants

By default an **enum** is the size of the smallest possible type (**char, short, int,** or **long**) that will contain all of the **enum** constants. However, the **-p0e** compiler option may be used to cause all **enum** constants to be represented as **int**s.

Multibyte Characters and Wide Characters

In order to support character sets that contain more characters that can be represented in an 8-bit **char**, ANSI C introduced wide characters and multibyte characters:

- A wide character is the internal representation of an element of such a character set. It has the type **wchar_t** which is defined in **stddef.h**

Internally, Aztec C68k/ROM represents wide characters using ASCII encodings. **wchar_t** is **typedef**'ed to be of type **char**.

- A multibyte character is the external representation of a wide character. It has the format of a normal C character string. Aztec C68k/ROM supports the standard American character set, thus multibyte characters are at most one character long.

size_t

The ANSI standard defines a type known as **size_t**, which is used by a number of library functions, and is also the return type of the **sizeof** operator. In the include files provided with Aztec C68k/ROM, **size_t** is **typedef**'ed to be of type **unsigned long**.

ptrdiff_t

The ANSI standard defines a type called **ptrdiff_t**, which is the type that results from subtracting two pointers. In the **stddef.h** include file provided with Aztec C68k/ROM, **ptrdiff_t** is defined to be a **long**.

Symbol Names

Symbol names are significant to 31 characters. This includes external symbols, which are significant to 31 characters throughout assembly and linkage.

A C name is converted to assembly language in one of several ways, as defined by the **-yu** option:

- An underscore can be prepended to the C name.(This is done by default)
- An underscore can be appended to the C name
- The assembler name can be the same as the C name.

Predefined Symbols

Aztec C68k/ROM pre-defines the following special symbols:

Symbol	Meaning
__FUNC__	Name of function being compiled
_INT32	Defined if ints are 32 bits long
_LARGE_CODE	Defined if module is compiled with **-mc**
_LARGE_DATA	Defined if module is compiled with **-md**
__VERSION	Set to the compiler version number
AZTEC_C	Defined to indicate that the compiler is Aztec

MCH_ROM	Defined to indicate that the C68k/ROM version of the Aztec compiler is being used

Register Variables

Aztec C68k/ROM lets you define the usage of registers for your programs. By default, registers are used as follows:

d2-d7/a3-a4	register variables
d0-d1/a0-a2	temporaries
a5	small model support registers
a6	frame pointer
a7	stack pointer

Register usage can be redefined using the **-y** options. Your program can explicitly allocate variables to registers, using the **register** keyword. You can also let the compiler assign variables to registers, using the **-sr** option.

The following types may be declared as register variables: **char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, pointer, float,** and **double**.

Register Function Calls

It is possible to use registers to pass arguments to regular functions. To do this use the **regcall** pragma:

#pragma regcall([*retreg***] *funcname* [(*regarg1, regarg2,...,regargn*)])**

where

retreg is the name of the register in which the function returns its value. If *retarg* is not specified, it defaults to d0.

regarg1, regarg2, ...,regargn are the names of the registers in which arguments are passed to the function.

funcname is the name of the function.

The following restrictions apply:

- 68881 registers can not be used for function arguments or return value.
- arguments and return values can not be of type **double, long double,** or **struct.**

Direct Functions

A DIRECT FUNCTION is a function that is assigned a sequence of numbers. When the compiler encounters a call to the function, it outputs the numbers instead of outputting code for a normal function call.

A direct function is defined as follows:

int *func0* **=** *val1;*

or

int *func0* **=** {*val1, val2, ...*};

where *func* is the name of the direct function and *val1, val2,...* are the numbers that are to be associated with *func.*

Each *val* is output in a two byte field.

Direct functions may be called like any other functions. The only difference is that the address of a direct function cannot be obtained.

The **-pd** option determines whether direct functions are supported or not; by default, they are supported.

In-line Assembly Code

Assembly language code can be interspersed within C source code by surrounding the assembly code with the statements **#asm** and **#endasm. For** example,

```
main()
{
/* C code */
#asm
; assembly language code
#endasm
/* C code */
}
```

Since **#asm** and **#endasm** are handled by the preprocessor, the compiler does not actually see the assembly code. Therefore, to assure that the correct code will be generated, it is a good idea to precede the **#asm** with a null statement (i.e., semicolon).

Variable names used within the C source may be referenced by name as follows:

- To access a global or static C variable or function from assembly language, you must use the same assembly language name that is used by the compiler generated code. The compiler translates a C name to assembler by either prepending an underscore to the C name, appending an underscore, or using the C name. By default, the compiler prepends underscores, but you can override this using the **-yu** option. For example, to

access a global variable whose C name is **count**, assembly code would use the name **_count** by default.

- To access a C function's argument and auto variables from assembly language, use the variables name with prepended %%. The following example illustrates this:

```
long total;
long sum (short, short)
void add five (short a)
{
      long result;
      short b;
      b=5:
#asm
      xref        _sum
      xref        _total
      move.w      %%b,-(sp)
      move.w      %%a,-(sp)
      jsr         _sum
      move.l      d0,_total
#endasm
}
```

For a discussion of register usage by assembly language programs, see the **Technical Information** chapter.

Pascal Functions

Using the **pascal** keyword for a function definition causes the following differences in the code generated when compared with a normal C function.

- When calling a pascal function, arguments are pushed on the stack in the order in which they're declared. This means the first argument is pushed on first rather than last as it is in C.

- The return value, if any, is returned on the stack rather than in a register. This requires that the caller allocate the appropriate amount of space on the stack for the return value which will be popped off when the function returns.

- The called pascal function cleans up the stack, removing any arguments that may have been passed on the stack by the caller. This differs from a C function where it's the responsibility of the calling function to clean up the stack.

For those writing code in C, these changes should be transparent. For those writing assembly code, it's essential to take these things into consideration when using the pascal specifier.

Pascal Character Strings

The format of a character string differs in C and Pascal. In C, the string consists of the characters with a terminating null character. In Pascal, the first byte of the string contains the number of characters in the string.

To have the compiler generate a Pascal format character string, begin the string with the sequence **\P** or **\p**. For example,

```
"\PThis is a Pascal string"
```

The string still will be null-terminated, so it can be passed to functions like **strcpy()** and **strcmp()**.

There are two functions that can be used to convert strings from one format to the other: **ctop()** converts a string from C to Pascal format, and **ptoc()** converts a string from Pascal to C format.

For more details on these functions, see their description in the **Library Functions** chapter of this manual.

Writing Machine-independent Code

The Aztec family of C compilers are almost entirely compatible. The degree of compatibility with v7 C, system 3 C, system 5 C, and Xenix C is also extrememly high. There are, however, some differences. The following paragraphs discuss things you should be aware of when writing C programs that will run in a variety of environments.

If you want to write C programs that will run on different machines, do not use bit fields or enumerated data types, and do not pass structures between functions. Some compilers support these features and some do not.

Compatibility Between Aztec Products

Within releases, code can be easily moved from one implementation of Aztec C to another. Where release numbers differ code is upward compatible, but some changes may be needed to move code down to a lower numbered release. The downward compatibility problems can be eliminated by not using new features of the higher numbered releases.

Sign Extensions For Character Variables

If the declaration of a **char** variable does not specify whether the variable is signed or unsigned, the code generated for some machines assumes that the variable is signed and others that it is unsigned. For example, none of the 8-bit implementations of Aztec C sign extend characters used in arithmetic computations, whereas all 16- and 32-bit implementations do. This incompatibility can

be corrected by declaring characters used in arithmetic computations as unsigned, or by AND'ing characters used in arithmetic expressions with 255(**0xff**). For instance:

```
char a=129
int b;
b = (a & 0xff) * 21;
```

The MPU... Symbols

To simplify the task of writing programs that must have some system dependent code, each of the Aztec C compilers defines a symbol which identifies the processor on which the compiler-generated code will run. These symbols, and their corresponding processors are:

Symbol	Processor
MPU 68000	68000/68008/68010/68020
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502/65C02
MPU8080	8080/8085
MPUZ80	Z80/HD64180
MCH_AMIGA	Amiga
MCH_MACINTOSH	Macintosh
MCH_ATARI_ST	Atari ST
MCH_ROM	68000 ROM system

Only one of these symbols will be defined for a particular compiler.

For example, the following program fragment contains several machine-dependent blocks of code. When the program is compiled for execution on a particular processor, just one of these blocks will be compiled: the one containing code for that processor.

```
#if MCH_MACINTOSH
  /*Macintosh code*/
#elif MPU8086
  /*8086 code*/
#elif MPU8080
  /*8080 code*/
#endif
```

Error Handling

There are two types of compiler errors—fatal and nonfatal. Fatal errors cause the compiler to make a final statement and stop. Running out of memory and finding no input are examples of fatal errors. The **Error Messages** chapter describes all these errors in detail. In addition, the nonfatal errors are also discussed there.

The compiler reports any errors it finds in the source file, displaying the source line in which the error was detected, with the underlined text highlighting the approximate point where the error occurred.

The compiler then displays a line containing the following information:

- name of the source file containing the line
- number of the line within the file
- an error code
- a message describing the error
- symbol that caused the error, when appropriate.

The compiler sends error messages to its standard output device. This can be redirected to a file in the normal way. Without the redirection of its standard output, the compiler sends error messages to the console. For example, to compile **prog.c** and send error messages to the file **prog.err**, use the following command:

```
c68 prog > prog.err
```

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error. If errors arise at compile time, you should first correct the first error, since this may clear up some of the errors that follow.

The best way to attack an error is first to look up the meaning of the error code in the **Error Messages** chapter. You will find hints there as to what the problem might be. And you will find it easier to understand the error and the message if you know why the compiler produced that particular code. The error codes indicate what the compiler was doing when the error was found.

Chapter 4 - Assembler

The Manx Aztec C Assembler, **as68**, translates assembly language source statements into relocatable object code. The assembler supports the instruction sets and addressing modes for the following chips: 68000, 68010, 68020, 68030, 68881, and 68851. By default, the assembler supports only the 68000; support for other chips is enabled using the **machine**, **mc68881**, and/or **mc68851** directives.

Assembler source statements are read from an input text file and the object code is written to an output file. A listing file is written if requested. The relocatable object code must be linked by the Aztec C Linker, **ln68**, before it can be executed. At linkage time it may be combined with other object files and runtime library routines from system or private libraries. Object modules produced from C source text and assembler source text can be combined at linkage time into a composite module.

Assembly language routines are generally not required when programming in C. Assembly language routines should only be necessary where critical execution time or critical size requirements exist. Some system interfacing or low level routines may also require assembler code. Even when assembler use is indicated, it may not be necessary to write separate assembler files, since the compiler supports in-line assembler code.

Information on the MC68000 and MC68020 architecture and instructions can be found in the Motorola *MC68000 16/32-bit Microprocessor Programmer's Reference Manual* and the Motorola *MC68020 32-bit Microprocessor User's Manual*, respectively (Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632).

This chapter is organized into the following sections:

- Operating Instructions
- Assembler Options
- Programmer Information

Operating Instructions

The assembler is started by entering the command line:

as68 [-*options*] *filename*

where [-*options*] specify optional parameters and *filename* is the name of the file to be assembled. (The compiler options **-a** or **-at** must be used when compiling to prevent the compiler from automatically invoking the assembler.)

The assembler reads assembly source statements from the input file, writes the translated relocatable object code to an output file, and if requested writes a listing to an output file. The assembler also merges assembly code from other files upon encountering an **include** directive.

Input File

The input file is a text file that is usually created by a text editor or the Manx Aztec C compiler. The input file resides in the current directory. If it does not, a fully qualified or partially qualified path name can be prefixed to the filename to designate the source directory. Although **.asm** is the recommended filename extension, any extension is acceptable.

The specification:

```
as68 x
```

assembles the file **x.asm** if it exists. If the file **x.asm** does not exist then the file **x** is assembled. If the file **x** does not exist or if file **x** cannot be assembled, then error messages will be returned.

Object Code File

The assembler writes the object code it produces to a file. By default this file is placed in the directory that contains the source file, and its name is derived from that of the input file by changing the extension to .r.

To write the object code to a file in another directory, and/or to a file having another name, use option **-o**.

For example, the following command assembles the source in **prog.asm**, sending the object code to the file **new.obj**. The latter file is placed in the current directory, because option **-o** does not specify otherwise.

```
as68 -o new.obj prog.asm
```

Listing File

If you specify option **-l**, the assembler produces a listing file with the same root as the input file and a filename extension of **.lst**. The listing file displays the source statements and their machine language equivalent. The listing also indicates the relative displacement of each machine instruction.

Optimizations

By default, the assembler performs some optimizations on an assembly language source file, by making two passes through the assembly source file.

Option **-n** disables some optimization, thereby allowing the assembler to run faster because it makes only a single pass through the source and because it does not optimize the code. However, usage of the **-n** option makes the resultant code larger and slower.

Optimizations affect the following instructions:

branches	converts long branches to short if possible and deletes branches to the following location
movem	If there are no registers, deletes the instruction. If there is only one register, substitutes the shorter **move** instruction. Note that the **move** instruction alters condition codes, while **movem** does not.
jsr	substitutes **bsr**, if possible.
jmp	substitutes **bra** if possible.

Searching For Include Files

By default the assembler searches only the current directory for files specified in **include** statements. Using the **-i** option and the INCL68 environment, you can make the assembler also search other directories, thus allowing source files and include files to be contained in different directories.

If the file name on the **include** directive specifies a directory or a drive, the assembler will search just the specified area for the file.

Option -i

Option **-i** defines a single directory to be searched for a file specified in an **include** statement. The path descriptor follows **-i** with no intervening blanks. For example, the specification

```
as68 -ie:\db\include prog
```

directs the assembler to search the `e:\db\include prog` area when looking for an include file. If desired, more than one **-i** may be used, thus defining multiple directories to be searched.

INCL68 Environment Variable

The **INCL68** environment variable, if it exists, also defines directories to be searched for include files.

The **INCL68** variable consists of the names of the directories to be searched, separated by semi-colons. For example, the following command creates the **INCL68** environment variable, defining three directories to be searched:

```
set INCL68=e:\;d:\project\include;c:\az68\include
```

These directories are (1) the root directory on the **e:** drive, (2) `c:\az68\include` the `\project\include` directory on the **d:** drive; and (3) the `\az68\include` directory on the **c:** drive

Include Search Order

When the assembler encounters an **include** statement, it searches directories for the file specified in the statement in the following order:

(1) current directory

(2) directories specified in option **-i** in the order listed on the line that started the assembler

(3) directories specified in the **INCL68** environment variable in the order listed

Assembler Options

Following is a list of assembler options. Detailed descriptions of these options are provided later in this chapter.

-a	Force total size of code and data to be aligned to a 4-byte boundary instead of the default 2-byte boundary.
-c	Make large code the default code memory model. May be overridden by the **near code** and/or **far code** directives. For details on memory models, see the **Compiler** chapter.
-d	Make large data the default data memory model. May be overridden by the **near data** and/or **far data** directives. For details on memory models, see the **Compiler** chapter.
-e*name*[=*val*]	Create an entry in the symbol table for *name* and assigns it the constant value *val*. If *val* is not specified, *name* is assigned the value 1.
-i*area*	Define an area to be searched for files specified in an **include** statement.
-l	Generate an assembly listing file with a **.lst** extension.
-n	Do not optimize object code.
-o *filename*	Specify name of output object module.
-v	Verbose option. Generate memory usage statistics.
-ZAP	Used primarily by the Aztec C compiler to direct the assembler to delete the input file after processing.

Programmer Information

There are four types of Assembler statements, all of which are discussed in the following sections:

(1) Comments
(2) Executable Instructions
(3) Directives
(4) Macro Calls

Comments

A comment can appear after a semicolon or after the operand field. For example:

```
; this is a comment
  link a6,#.2    ;this is also a comment
```

A line whose first non-blank character is an asterisk or a semicolon is assumed to be a comment and is ignored by the assembler.

Executable Instructions

Executable instructions have the general format:

label operation operand comment

Labels

Assembler labels can be any length. External labels are only significant for the first 31 characters. Any additional characters are ignored. Valid label characters include letters, numbers, or the special characters "." and "_". A label cannot begin with a digit unless it is a temporary label. Labels that do not start in the first column require a colon suffix.

Temporary Labels

Temporary labels of the form *n*$, where *n* consists of decimal digits, are supported. These labels are in effect until the the next nontemporary label is encountered. For example:

```
1$  move.l      (a0)+,(a1)+
    dbra        d0,1$
```

Operations

The assembler recognizes all of the mnemonics found in Motorola's reference manuals for the 68000, 68020 processors and the 68881, 68851 coprocessors

To specify a length for instructions that support multiple lengths, suffix the instruction mnemonic with:

.B	8-bit operands
.W	16-bit operands
.L	32-bit operands

Operands

The operand field consists of one expression, or two expressions separated by a comma with no embedded spaces. An expression is comprised of register mnemonics, symbols, constants, or arithmetic combinations of symbols or constants. In addition certain 68020 instructions require special operand syntax.

Symbols

Symbols or labels represent relocatable or absolute values. An absolute value is one whose value is known at assembly time. A relocatable value is one whose value is not known until the program is actually loaded into memory for execution.

Relocatable expressions can only be expressed arithmetically as sums or differences. The difference between two relocatable expressions is absolute. The result of summing two relocatable expressions is undefined.

Constants

There are five types of constants: octal, binary, decimal, hexadecimal and string. An octal constant is expressed as an @@ followed by a string of digits from the set 0 through 7 such as `@@777`.

A binary constant is expressed as a % followed by a string of ones and zeroes, such as

`%10101` or `%11001100`

A decimal constant is a string of numbers. A hexadecimal constant is a $ followed by a string of characters made up of numbers or letters from a through f such as

`$ffff` or `$1a2e`

A string constant is any string of characters enclosed in single quotes such as

`'abdc'`

Registers

Register mnemonics are:

Name	Register
d0, ..., d7	Data registers
a0, ..., a7	Address registers
sp or a7	Stack pointer
pc	Program counter
sr	Status register
ccr	Condition code register
usp	User stack pointer

Operand Expressions

The assembler supports operand expressions that use the following operators:

Operator	Meaning
+	Addition
-	Subtraction & unary minus
*	Multiplication
/	Division
>>	Shift right
<<	Shift left
&	And
|	Or
!	Inclusive or
^	Exclusive or
~	Bitwise not
//	Modulo

The order of precedence is innermost parenthesis, unary minus, shift, and/or, multiplication/division, addition/subtraction, bitwise, and logicals.

Instruction Comments

A line that contains an instruction or directive can also contain comments; in this case, the comments are placed after the operands of the instruction or directive.The assembler can be in either of two modes for determining where a comment begins on a line that also contains an instruction or a directive.

- The first blank character encountered in the operand field signifies the end of the operands and the beginning of the comments. Thus, in this mode, the operand field cannot contain embedded blanks. This mode, which is compatible with the Motorola assembler, is the default mode.

- A semicolon signifies the beginning of the comments.

The mode is selected using the **blanks** directive.

Directives

The following paragraphs describe the directives that are supported by the assembler.

blanks

blanks {on | off}
blanks {yes | no}
blanks {y | n}

This directive controls whether the assembler accepts blanks or tabs in the operand field of the instruction.

Setting **blanks off** treats a blank as the end of the operand field. This is the default setting.

blanks on allows blanks to be placed between any two complete items. With this setting all comments must be preceded by a ;.

clist and noclist

These directives specify whether or not statements should be included in the listing file, when the statements were not assembled as a result of assembler conditional statements. By default, such statements are not listed.

cnop

label cnop *n1,n2*

This directive is used to force alignment on any boundary at a particular offset.

The first value, *n1*, is an offset while the second value, *n2*, specifies the alignment to be used as the base of the offset. For example, to align to an even word boundary:

```
cnop  0,2
```

while to align to a long word boundary:

```
cnop  0,4
```

and finally to align to a word beyond a long word boundary:

```
cnop  2,4
```

Note that this will only take effect relative to the beginning of the current module's code or data. Normally, the linker will not align individual modules to long word boundaries. So, for this directive to work, it must be the first module linked into the program, or the linker's **-a** option must be used to force long word alignment of modules.

cseg

The code following this directive is placed in the program's code segment.

dseg

The code following this directive is placed in the program's initialized data segment.

dc - Define Constant

[*label*]	dc.b	*value*[,*value*, *value* ...]
[*label*]	dc	*value*[,*value*, *value* ...]
[*label*]	dc.w	*value*[,*value*, *value* ...]
[*label*]	dc.l	*value*[,*value*, *value* ...]
[*label*]	dc.b	*"string"*

The **dc** directive causes one or more fields of memory to be allocated and initialized. Each *value* operand causes one field to be allocated and then to be initialized with the specified value. A *value* can be an expression. An expression may contain forward references.

Each field for a particular **dc** directive is the same length. A period followed by **b**, **w**, or **l** can be appended to a directive, defining the field length to be one, two, or four bytes, respectively.

If the field length is not specified in this way, it defaults to two bytes.

Fields that are two or four bytes long are aligned on word boundaries.

The last form listed for **dc** allocates a field having exactly the number of characters in the string, and places the string in it. **Note**: Trailing null characters must be explicitly requested, for example

```
dc.b "Hello",0.
```

dcb - Define Constant Block

[*label*]	dcb.b	*size*[,*value*]
[*label*]	dcb	*size*[,*value*]
[*label*]	dcb.w	*size*[,*value*]
[*label*]	dcb.l	*size*[,*value*]

The **dcb** directive allocates a block of storage containing *size* fields, and initializes each field with *value*. If *value* is not specified, it is assumed to be 0.

Each field for a particular **dcb** directive is the same length. A period followed by **b**, **w**, or **l** can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length is not specified in this way, it defaults to 2 bytes (**.w**).

Fields that are two or four bytes long are aligned on word boundaries.

ds - Define Storage

[*label*] ds.b *size*
[*label*] ds *size*
[*label*] ds.w *size*
[*label*] ds.l *size*

This directive allocates a block of storage containing *size* fields, and sets each field to 0.

Each field for a particular **ds** directive is the same length. A period followed by **b**, **w**, or **l** can be appended to a directive, defining the field length to be one, two, or four bytes, respectively.

If the field length is not specified in this way, it defaults to 2 bytes(**.w**).

Fields that are two or four bytes long are aligned on word boundaries.

entry

[*label*] entry *symbol*

This directive defines the entry point of the program. Only one entry can be declared per program. If no entry point is defined, the first instruction of the first module becomes the default entry point.

end

This directive defines the end of the source statements. All files are closed and the assembler terminates.

equ

label* equ *expression

This directive assigns the value of the expression on the right to the label on the left.

equr

label **equr** *register*

This directive allows a register to be referenced by an alternate name. Reference to the new name is made without regard to case.

even

[*label*] *even*

This directive forces alignment to a word (16 bit) boundary.

fail

fail

This directive causes the assembler to generate an error for this line. This can be used in macros which detect an incorrect number of arguments and wish to prevent assembly.

far code

[*label*] **far code**

This directive causes the assembler to generate code for the large code memory model. Absolute addressing will be used. If this directive is not specified, the memory model used will be determined by the presence or absence of the **-c** assembler option.

far data

[*label*] **far data**

This directive causes the assembler to generate code for the large data memory model. Absolute addressing will be used. If this directive is not specified, the memory model used will be determined by the presence or absences of **-d** assembler option.

freg

label **freg** *register list*

This directive is like the **reg** directive, except that it is used to specify the floating point registers of the MC68881. The list is either composed of the floating point registers **fp0** through **fp7** or of the floating point control registers **fpcr, fpsr,** and **fpiar,** but not both.

ifc and ifnc

ifc *'string1','string2'*
ifnc *'string1','string2'*

These conditionals check to see if the two strings are equal. If they are, **ifc** will assemble the following code, while **ifnc** will skip it.

ifd and ifnd

ifd *symbol*
ifnd *symbol*

These conditionals check to see if the specified symbol has been defined or not. If *symbol* has been defined, then **ifd** will assemble the following code, while **ifnd** will not.

if, else, and endc

if *test*
...
[else]
...
endc

These directives are used to allow conditional assembly of parts of the input file. The general form of the **if** test is:

exp
exp == *exp* || *exp* = *exp*
exp != *exp* || *exp* <> *exp*
'str1' == *'str2'* || *'str1'* = *'str2'*
'str1' != *'str2'* || *'str1'* *'str2'*

If the test result is true, then the lines up to an **else** or **endc** are assembled. If there is an **else**, then lines up to the **endc** are skipped. The skipped lines are not displayed in the listing file unless the **clist** directive has been used. If the test is false, then lines are skipped until an **else** or **endc** is encountered. If it is an **else** then the following lines up to an **endc** are assembled. An undefined symbol is treated as having the value 0.

near code

[*label***]** **near code**

This directive causes the assembler to generate code for the small code memory model. It is the default unless the **-c** option, or **far code** directive, is used.

near data

[*label*] near data

This directive causes the assembler to generate code for the small data memory model. It is the default unless the **-d** option, or **far data** directive, is used.

other ifs

ifeq *absolute_expression*
ifgt *absolute_expression*
ifge *absolute_expression*
ifle *absolute_expression*
iflt *absolute_expression*
ifne *absolute_expression*

These conditionals perform a comparison of the value of the absolute expression to zero. If the specified condition is true, then the following assembly language is processed, otherwise it is skipped.

include

include *'filename'*
include *"filename"*
include *<filename>*

This directive causes the contents of the file specified to be processed by the assembler as if they had appeared at the same relative location as the include statement.

global and bss

[*label*] global *symbol, size*
[*label*] bss *symbol, size*

These directives reserve storage for uninitialized data items. The area is reserved in the uninitialized data area. If **global** is used then the data item is known to other modules that are external to the routine. If **bss** is used then the data item is local to the routine in which it is defined.

If a **global** is defined in more than one module, the linker will generate the error message:

```
each external name must have exactly one definition
```

A symbol that appears in both a **global** and a **public** directive is located in the initialized data area and the global statements *size* parameters are ignored.

list and nolist

The directives **list** and **nolist** turn on and off, respectively, the listing of assembly language statements to the listing file.

mlist and nomlist

The directives **mlist** and **nomlist** specify if the assembly language statements generated by a macro expansion should be written to the listing file.

machine

machine mc68000
machine mc68010
machine mc68020

This directive enables or disables the additional instructions and addressing modes associated with different processors in the MC68000 family.

macro and endm

macro *symbol*
...
text
...
endm

or

symbol **macro**
...
text
...
endm

symbol is entered in the assembler opcodes table. The text between the **macro** and **endm** is saved in memory. When *symbol* is encountered as an opcode the text is placed in line.

Up to nine arguments can be specified in one of two ways. There is also an argument **0** which refers to the extension on the **macro** directive when it was invoked. The arguments are referenced

in the macro text as either %0 through %9 or \0 through \9. In expanding a macro symbolic argument, references are replaced by their actual value.

The assembler also has facilities for generating unique labels within a macro. When the \@ is specified within a macro, the assembler will generate labels of the form *.nnn* where *nnn* will have a unique value for each invocation of the macro.

The symbol *narg* is a special assembler symbol that indicates the number of arguments specified when the macro is invoked. Outside of macro definitions, the value of *narg* is **0**.

Macro arguments that contain a space or comma can be enclosed in bracketing "<" and ">" characters.

mc68851

This directive enables the **mc68851** memory management instructions to be recognized and assembled by the assembler.

mc68881

This directive enables the **mc68881** floating point instructions to be recognized and assembled by the assembler.

mexit

Upon encountering this directive, expansion of the current macro stops and the assembler scans for the statement following the **endm** directive.

public

[*label*] **public** *symbol1*[,*symbol2*..]

This directive identifies the specified symbols as having external scope. These symbols are visible to the linker and are used to resolve references between modules. The type of the symbol is CODE if it appears within the code segment and DATA if it appears within the data segment, and ABS if it was defined to have an absolute value in an **equ** directive.

reg

label **reg** *register list*

This directive assigns the value of the register list to the label. Forward references are not allowed. *register list* consists of a list of register names separated by the / character. The - character may be

used to identify an inclusive set of registers. This directive is generally used with the **movem** instruction.

The following are valid register lists:

a0-a3/d0-d2/d4
a1/a2/a4/a6/a0-a2

section

[*label*] section *name*, code
[*label*] section *name*, data
[*label*] section *name*, bss

This directive performs the same functions as the **cseg** and **dseg** directives. The *name* parameter, if present, is ignored at the current time. The *type* parameter is used to switch from **code** and back again. If only a *name* parameter is specified, the *type* defaults to **code.**

set

label* set *expression

This directive assigns the value of the absolute *expression* to the symbol specified by *label*. This definition is similar to the **equ** directive, with the exception that this symbol's value can be changed with another **set** directive. This includes expressions referencing the current value of the symbol itself. For example:

```
sym     set   sym+1
```

takes the current value of **sym** and adds **1** to it, which then becomes the *new* value of **sym**.

ttl

ttl *title_string*

This directive sets the title of the current module being assembled. This directive is implemented for compatibility with other assemblers and has no effect at the current time.

xdef and xref

xdef *symbol*
xref *symbol*

These directives are used to specify the definition and reference of global symbols.

Macro Calls

Macro calls consist of a macro name with an optional label, extension, and arguments, in this form:

[*label*] *macroname*[*.ext*] [*arg1, arg2...*]

The optional extension consists of a "." followed by any valid 680X0 extension, such as w, l, etc.

Up to nine arguments may be passed to the macro.

See the "macro and endm" section of this chapter for more information regarding macros.

LINKER 5

Chapter 5 - Linker

This chapter describes the **ln68** linker. It first gives a brief introduction to linking; the second and third sections give detailed operator-type information about the linker; and the fourth section gives programmer-type information.

Introduction to Linking

C encourages modular programming; that is, the partitioning of a program into source modules that are separately compiled and assembled. The compilation and assembly of a source module generates an "object module". The linker links together all of a program's object modules, creating an executable program.

Programs typically consist of many object modules. Since it would be inconvenient to explicitly specify each module whenever you link a program, Aztec C68k/ROM supports object module libraries. When you pass a library's name to the linker, it examines the library's modules, and links into the program just those that are needed.

Aztec C68k/ROM provides source for several frequently-used functions, and for support routines that are called by compiler-generated code to perform operations such as arithmetic computation, etc. These are in source form, and part of the process of installing Aztec C68k/ROM is to compile and assemble them and then create object module libraries of them. In the following discussion, we refer to one of these libraries, **c.lib** , which contains non-floating point functions, and whose modules have been compiled to use the small code, small data memory model.

Some of the provided functions, called "standard I/O" functions, perform high-level I/O by calling functions that you must write, as described in the **Library Customization** chapter. In the following discussion, we assume that you have implemented these functions, and thus that your **c.lib** library supports the standard I/O function **printf()**.

Creating the 'hello, world' Program

Let's consider the creation of the **hello, world** program, whose main module, in the file **hello.c** , looks like this:

```
main()
{
   printf("hello, world\n");
}
```

The object modules that must be linked together include **hello.r**, the **printf()** module from **c.lib** , and other "support" modules from **c.lib**. You don't explicitly generate calls to these support modules; they're automatically generated by the compiler. The command to link the program is:

```
ln68 hello.r -lc
```

The **hello.r** operand causes the linker to include **hello.r** in the program. The **-lc** operand causes the linker to search for needed modules in the **c.lib** library that's located in the directory specified by the INCL68 environment variable and to include them in the program.

Another Example

As another example, consider a program consisting of two of your own modules, plus whatever modules are needed from **c.lib**. The source for the first of these modules, **file1.c**, looks like this:

```
main()
{
   printf("second example");
   func1();
   func2();
}
func1()
{
   return;
}
```

The source for the second module, **file2.c**, looks like this:

```
func2()
{
   return;
}
```

The command to link this program is:

```
ln68 file1.r file2.r -lc
```

This causes the linker to include object modules **file1.r** and **file2.r** in the program, and to search for other needed modules in **c.lib**.

Symbol Reference and Definition

As the linker proceeds, it keeps track of the global symbols that each module references and defines. For the linkage to succeed, each symbol that's referenced must also be defined; there can be multiple references to the same symbol.

Here are some examples of symbol reference and definition:

- A call to a function is a reference to that function's name;
- The actual definition of a function is a definition of the function's name;
- **extern** keyword is a definition of the variable.
- A variable declaration that includes the **extern** keyword is a reference to the variable.

- A global declaration of a variable that doesn't include the **extern** keyword is a definition of the variable.

For example, in the above sample program, **file1** contains references to **printf()**, **func1**, and **func2**, and to support routines; it contains definitions of **main** and **func1**. **file2** contains a definition of **func2**, and references to support routines. Within **c.lib** are modules that define **printf()** and the support routines.

When the linker has examined all the modules that are going to be linked into a program, it checks its lists of defined and referenced symbols. If there are symbols that are referenced but not defined, the linker issues messages saying that those symbols are undefined and then halts without completing the linkage. For example, if the link command for the above program specified just **file1.r**, the linker would issue a message saying that **printf()**, **func2**, and the support routines were undefined, since the references to those symbols were not matched by definitions. It doesn't say that **func1** is undefined, because the reference to it is matched by its definition in the same file.

Searching Libraries

When the linker is searching a library, it checks each module's defined code symbols (i.e., symbols that are defined in the module's code segment), looking for symbols that have been been referenced but not defined in the modules that have already been included in the program. If it finds such a symbol, it includes the module that contains it in the program. For example, in the above linkage the symbol **printf()** is referenced but not defined when the linker begins searching **c.lib**. When the linker looks at the library's module that contains the definition of the **printf()** code symbol, the linker includes that module in the program it's building.

It's important to note that only the definition of a code segment symbol in a library module can cause the linker to include the module in a program. For example, in the above linkage the definition of a **printf()** data symbol (i.e., a symbol located in the data segment) in a library module would not cause the linker to include that module in the program.

The Ordering of Module and Library Names

The order in which modules and libraries are specified on the command line is important, since the linker processes files in this order.

For example, an attempt to link the **hello, world** program with the following command will fail:

```
ln68 -lc hello.r
```

For this command, the linker first scans **c.lib** and then **hello.r**. When it scans **c.lib** there aren't yet any referenced but undefined symbols, so the linker won't include any of the library's modules in the program. When it includes **hello.r** in the program, **printf()** and the referenced support routines become referenced but undefined. But since **hello.r** is the last module

specified on the command line, the linker won't go back and rescan **c.lib** ; so the undefined symbols remain undefined, and the linkage fails.

The moral of this is that it's good practice to leave all libraries at the end of the command line, with **c.lib** at the very end.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant, because the specification of a library on the command line causes the linker to search that library just once, from beginning to end. If a module is pulled in at any point, and that module introduces a new undefined symbol, then that symbol is added to the running list of undefined's. The linker will not search the library twice to find definitions for unmatched references.

For example, suppose you have a program that contains the modules **main.r**, **input.r**, **calc.r**, **output.r**, and any needed library modules, and that your modules have the following references:

Module	Definitions	References
main.r	main	in, calc
input.r	in	gets
calc.r	calc	out
output.r	out	printf

The command to link the program would look like this:

```
ln68 main.r input.r calc.r output.r -lc
```

Suppose we build a library, **sub.lib** , to hold the last three modules of this program. Then our link step will look like this:

```
ln68 main.r -lsub -lc
```

The order of the modules in **sub.lib** is important. For example, suppose **sub.lib** 's modules are in the following order:

```
input.r
output.r
calc.r
```

With the library in this order, here's how the above linkage would proceed:

- The linker includes **main.r** in the program. After this step, **in** and **calc** are referenced but undefined (as are some other symbols that are in **c.lib**, but we're not concerned about them right now).

- The linker begins searching **sub.lib**, and looks first at its **input** module. Since that module defines **in**, which is one of the linker's referenced but undefined symbols, it includes the **input** module in the program, takes **in** off its list of referenced but undefined symbols, and adds **gets** to it.

- The linker looks at **output**, the next module in **sub.lib**. At this point, The symbols **calc** and **gets** are referenced but undefined. Since neither of these symbols are defined in **output**, the linker ignores it

- The linker looks at **calc**, the next and last module in **sub.lib**. Since this module contains a definition of **calc**, one of the linker's referenced but undefined symbols, the linker includes **calc** in the program, removes **calc** from its list of referenced but undefined symbols, and adds **out** to the list.

- The linker next scans **c.lib**, and includes the modules within it that define **gets** and the support routines.

After scanning all of these modules and libraries, the **out** symbol is still referenced but undefined, so the linker will abort after logging the following message:

```
Undefined symbol: _out
```

This means that the module defining **out** was not pulled into the linkage. The reason, as we saw, was that **out** was not a referenced symbol when the linker scanned the **output** module, so the linker ignored it.

This problem would not occur if **sub.lib** 's modules were in the following order:

```
input.r
calc.r
output.r
```

The ord68 Library Utility

The **ord68** utility simplifies the task of creating a library, by sorting a list of names of files that contain object modules. A library of these object modules that is created using the sorted list will be in the correct order.

There are some sets of object modules whose modules can't be put in a "correct" order; that is, for which it is impossible for the linker to decide which of the library's modules are needed by making just a single scan through the library. For such libraries, you can explicitly tell the linker to search the library multiple times.

For example, if **sub.lib** required two passes to find all needed modules, you could link the above program using the command.

```
ln68 main.r -lsub -lsub -lc
```

Using the Linker

The command to link a program looks like this:

ln68 [-options] *file1.r [file2.r ...] [lib1.lib2 ...]*

where *-options* are special options, *file1.r, file2.r* are names of the object modules that are to be included in the program, and *lib1.lib2 , ...* are names of the libraries that are to be searched for needed modules. The object modules must have been created using **as68** and the libraries by **lb68.**

The Executable File

You can specify the name of the file to which the executable program is written with the **-o** linker option. Otherwise, the linker will derive the name of the output file from that of the first object module file listed on the command line, by deleting its extension. In the default case, the executable file will be located in the directory in which the first object file is located. For example,

```
ln68 prog.r -lc
```

will produce the file **prog**, by linking the object module **prog . r** together with needed modules from the library **c . lib** . (The **-l** option provides a convenient means of specifying libraries, as discussed below).

A different output file can be specified with the **-o** option, as in the following command:

```
ln68 -o prog mod1 . r mod2 . r -lc
```

Libraries

The following libraries are provided with Aztec C68k/ROM:

- **c.lib** and variants, each of which contain functions that perform non-floating and internal functions that are called by compiler generated code.
- **m.lib** and variants, each of which contain functions that perform floating point operations using software.
- **m8.lib** and variants, each of which contain functions that perform floating point functions using the 68881.

Each of the versions of a given library has been compiled using a different combination of integer size and memory model.

All programs must be linked with the version of **c.lib** that matches the **int** size and memory model used by your modules.

Programs that perform floating point must be linked with the appropriate version of **m.lib** or **m8.lib**. The floating point library must be specified on the command line before **c.lib**.

Libraries of your own modules can also be searched by the linker. These are created with the Manx **lb68** program, and must be listed on the linker command line before the Aztec libraries.

For example, the following links the module **prog.r**, searching the libraries **mylib.lib**, **new.lib**, **m.lib**, and **c.lib** for needed modules:

```
ln68 program.r mylib.lib new.lib -lm -lc
```

Each of the libraries will be searched once in the order in which they appear on the command line.

Linker Options

Summary of Options

-o *file*	Write executable code to the file named *file*.
-l*name*	Search the library *name*.lib for needed modules.
-f *file*	Read command arguments from *file* .
-g	Start collecting source level debugging information. (See also **-q**)
-m	Inhibit warnings of user symbols overriding library symbols.
-q	Stop collecting source level debugging information. (See also **-g**)
-t	Generate an ASCII symbol table file.
-v	Be verbose (i.e. list detailed information about each segment).
+r *n*	Use address register *n* for small model operations. *n* is a decimal value, and defaults to 5 (ie, address register **a5**).
+c *xxxx*	Set origin of code section to the hex value *xxxx* (default: 0).
+d *xxxx*	Set origin of initialized data section to the hex value *xxxx* (default: immediately after the code section).
+u *xxxx*	Set origin of the uninitialized data section to the hex value *xxxx* (default: immediately after the initialized data section).
+s *xxxx*	Set the size of the stack area to the hex value *xxxx* (default: 2k).
+j *xxxx*	Set the program's initial stack pointer to the hex value *xxxx* . (default: stack area immediately follows uninitialized data section, with size specified by **+s** option; stack pointer points to the top of this area).
+a	Toggle 'long align' mode. When this mode is enabled, each module's code begins on a longword boundary; i.e. on a byte whose address is a multiple of 4. By default, this mode is disabled.
+q	Be quiet; i.e. don't list, on the console, each module that is included in a program. By default, the linker issues this list.

Detailed description of the options

The -o Option

The **-o** option can be used to specify the name of the file to which the linker is to write the executable program. The name of this file is in the parameter that follows the **-o** . For example, the following command writes the executable program to the file **progout** :

```
ln68 -o progout prog.r -lc
```

If this option isn't used, the linker derives the name of the executable file from that of the first input file, by deleting its extension.

The -l Option

The -l option provides a convenient means of specifying to the linker a library that it should search, when the library is in a directory identified by the INCL68 environment variable, and when the extension of the library is .lib .

The name of the library is derived by concatenating the value of the environment variable CLIB68, the letters that immediately follow the **-l** option, and the string **.lib** . For example, with the libraries **subs . lib**, **io . lib**, **m . lib** , and **c . lib** in a directory specified by CLIB68 , you can link the module **prog . r**, and have the linker search the libraries for needed modules by entering

```
ln68 prog.r -lsubs -lio -lm -lc
```

The -f Option

-f *file* causes the linker to read command line arguments from *file*. When done, it continues reading arguments from the command line. For example, the following command causes the linker to create an executable program in the file **myprog**. The linker includes the modules **myprog . r**, **mod1 . r**, and **mod2 . r** in the program, and searches the libraries **my . lib** and **c . lib** for needed modules.

```
ln68 myprog.r -f myprog.lnk -lc
```

where the file **myprog . lnk** contains the following:

```
mod1.r mod2.r
mylib
```

The linker arguments in a **-f** *file* can be separated by tabs, spaces, or newlines.

There are several uses for the -f option. The most obvious is to supply the names of modules that are frequently linked together. Since all the modules named are automatically pulled into the linkage, the linker does not spend any time in searching, as with a library. Furthermore, any

linker option except **-f** can be given in a **-f** *file*. **-f** can appear on the command line more than once, and in any order. The arguments are processed in the order in which they are read, as always.

The -g and the -q Option

The **-g** option causes the linker to start or resume the collection of source-level debugging information for the program that is being linked.

The **-q** option causes the linker to stop the collection of source-level debugging information.

These two options can be used to selectively collect source-level debugginf information about specific modules. For example, the following command tells the linker to only collect debugging information for the **prog** and **sub2** modules:

```
ln68 -g main -q sub1 -g sub2 -q -lc
```

Inorder for the linker to collect debugging information about a module, the module must have been compiled with the **-bs** option.

The linker writes the source-level debugging information that it collects to a file. The name of this file is derived from the name of the file to which the linker writes the executable program, by changing the extension to **.dbg**.

The -m Option

Normally, when you create a variable in your program that has the same name as a library routine, the linker issues a warning that your symbol will override the library symbol. If the **-m** option is specified, this warning is suppressed.

The -t Option

The **-t** option causes the linker to write the program's symbol table to a file. This file lists each of the program's symbols and its address. The file is organized into four sections:

- Symbols in the code section (preceded by the line "Segment: 00 Hunk: 00);
- Symbols in the initialized data section (preceded by the line "Segment: 00 Hunk: 01);
- Symbols in the uninitialized data section, (preceded by the line "Segment: 00 Hunk: 02);
- Values of the program's constant symbols (STKSIZ is the size of the program's stack area, and **_stkorg** is the initial stack pointer).

The symbol table file will have the same name as that of the file containing the executable program, with extension changed to **.sym**.

There are several special symbols which will appear in the table. They are defined later in this chapter, in the "Programmer Information" section.

The +r Option

The **+r** option defines the address register that will be used in support of modules that use the small code and/or small data memory model. It has the format **+r** ***dd*** , where *dd* is the number of the address register.

For example, the following command tells the program to use address register **a4** as the small model support register:

```
ln68 +r 4 main.r -lc
```

If this option isn't specified, address register **a5** is used.

If any of a program's modules use small code and/or small data, the small model support register points into the program's data sections. When a small data module attempts to access a variable that's in a data section, the variable's address is specified as a displacement from the small model support register. When a small code module calls a function that is more than 32k away from the call instruction, the linker will generate a jump instruction to the target function, place the instruction in the program's data area, and change the PC-relative call to a call of the generated jump instruction; the converted call will specify the address of the jump instruction as a displacement from the small model support register.

For more information about memory models, see the "Programmer Information" section of the **Compiler** chapter.

Options for Positioning a Program's Sections

The linker organizes a program into three sections: code, initialized data, and uninitialized data. You can define the starting addresses of these sections using the **+c**, **+d**, and **+u** options; an option is followed by the hex value of the desired starting address.

By default, the code section begins at address 0, the initialized data section immediately after the code section, and the uninitialized data section immediately after the initialized.

For example, the following command creates the program **prog** whose code section begins at address **0**, initialized data at **0x8000**, and uninitialized data at **0x10000**:

```
ln68 +d 8000 +u 10000 prog.o -lc
```

Stack Options

Two options affect a program's stack: **+j** and **+s**. The **+j** option defines the location at which the program's stack register initially points. The address in hex of this location follows the **+j**. For ex-

ample, the following command creates a program whose stack register initially points at **0x20000**:

```
ln68 +j 20000 prog.r -lc
```

If the **+j** option isn't specified, the stack register will initially point to a location that follows the program's uninitialized data section. You can specify the distance between this location and the end of the uninitialized data section with the **+s** option. The hex value of the distance follows the **+s**. For example, the following command creates a program whose stack register initially points to a location that is **0x1000** bytes above the end of its uninitialized data section:

```
ln68 +s 1000 prog.r -lc
```

The default value of the **+s** option is 2k; this means that when you specify neither the **+s** nor **+j** options, the program's stack register will point to a location that is 2k bytes beyond the end of its uninitialized data section.

The linker creates two stack-related symbols: **_Storg_** , whose value is the address initially pointed at by the linked program's stack register; and **STKSIZ** , whose value is the explicitly- or implicitly-defined value of the **+s** option. The standard startup routine uses **_Storg_** to set up the stack register; it doesn't use **STKSIZ** .

Programmer Information

This section contains bits of information about the linker that you may find useful.

Program Format

The linker creates a program that's in CP/M-68k format, with no relocation records.

Special Linker-created Symbols

When the linker creates a program, it defines several global symbols. These are:

_H0_org and **_H0_end**	Beginning and ending addresses of program's code section.
_H1_org and **_H1_end**	Beginning and ending addresses of program's initialized data section.
_H2_org and **_H2_end**	Beginning and ending addresses of program's uninitialized data section.
Storg	Initial contents of program's stack pointer.
STKSIZ	Size of program's stack area (used when **+j** option isn't used).

Entry Points

The ENTRY POINT for a program is the location at which execution of the program is to begin. The entry point is defined using the assembly language entry directive.

To allow a program's entry point to be located somewhere in the middle of the program, the linker supports the following feature; if the program has an entry point that is not at the beginning of the program's code segment, the linker will create a jump to the entry point at the beginning of the code segment.

For example, consider what happens when you create the **hello world** program using the following commands:

```
c68 hello.c
ln68 hello.r -lc
```

This program's startup code is in the module **rom68.r** within **c.lib**, and its entry point is the symbol **.begin** within this module. This module is located in the middle of the program since **hello.r** is the first module of the code segment. Thus, the linker automatically creates an instruction at the beginning of the program's code segment that jumps to **.begin**.

You can prevent the linker from automatically creating this jump instruction to the program's entry point by doing one of the following.

- Remove the **entry** directive from the program's modules; or
- Link the program so that its entry point is at the beginning of the code segment.

For example, if you have a startup module named **startup.r** that you want linked into the **hello world** program instead of **c.lib**'s **rom68** module, you could link the program using the following command:

```
ln68 startup.r hello.r -lc
```

If **startup.r** has an entry point that is not at the beginning of the module's code, the linker will create a jump instruction to the entry point and put it at the beginning of the program's code segment. If **startup.r** either does not have an entry point, or the entry point is at the beginning of the module's code segment, the linker will not create a jump instruction.

The entry point in **c.lib**'s **rom68** startup module is named **.begin**. The entry point is not required to have this name, but it usually is given this name. Here's why:

- When a module is compiled, the compiler by default generates a reference to **.begin**, even though the module being compiled does not reference this symbol (the **-m0b** option prevents the compiler from generating the reference to **.begin**)
- When the program is linked, the compiler generated references to **.begin** force the linker to include **c.lib**'s **rom68** startup module, unless you have linked in a module that defines **.begin**.

The presence of an **entry** directive in a library module does not cause the linker to include the module in a program; it just identifies the symbol as being the entry point, in case the linker needs to create a jump instruction at the beginning of the program's code segment to the entry point.

Chapter 6 - Utilities

This chapter describes the Aztec C utility programs, in alphabetical order. Here is a list of those utilities along with brief descriptions:

arcv	dearchive **mkarcv** created archive
cnm68	display object file info
coff68	convert program to **coff** format
crc	compute file checksums
diff	compare source files
grep	search source files
hd	generate hex dump for files
hex68	convert programs to Intel Hex
lb68	create and maintain object libraries
make	automatically keep program's libraries, etc. up-to-date
mkarcv	create source archive
obd68	list object code
ord68	sort object module list
srec68	convert program to Motorola S-record
tekhex68	convert program to Tektronix Extended Tek Hex code

arcv source dearchiver

SYNOPSIS

arcv *arcvfile* [*dest_ dir*]

DESCRIPTION

arcv extracts the source from the archive file named *arcvfile* and places the result in separate files. *arcvfile* must have been created with **mkarcv**. The original archived file is left intact. **arcv** generates the name of the file it wants to create by prepending *dest_dir* to the file name as recorded in the archive. If a generated name contains path information (that is the directories that must be passed through to get to the file), **arcv** will create the directories that are in that path before creating the file.

EXAMPLE

For example, suppose you have an archived file named **example.arc** which contains two files, **explore.c** and **makefile**, then the command

```
arcv example.arc
```

will place the files **explore.c** and **makefile** in the current directory.

For example, suppose you have created, using **mkarcv**, a source archive named **test.arc** that contains the files **test.c** and **makefile** in the current directory, enter:

```
arcv test.arc
```

To create **test.c** and **makefile** on the current directory of the **b:** drive, enter:

```
arcv test.arc b:
```

To create **test.c** and **makefile** on the **\src** directory on the **c:** drive, enter:

```
arcv test.arc c:\src\
```

If **c:\src** does not exist, **arcv** will automatically create it. Note the terminating slash on **c:\src**. This is required because of the way **arcv** handles the names of the files that it creates, by prepending **c:\src** to the names **test.c** and **makefile**.

cnm68 display object file info

SYNOPSIS

cnm68 [*-options*] *file 1* [*file 2...*]

DESCRIPTION

cnm68 displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler or libraries of object modules created by the **lb68** librarian.

For example, the following displays the size and symbols for the object module **sub1.r** and the library **c.lib**:

```
cnm68 sub1.r c.lib
```

By default, the information is sent to the console. It can be redirected to a file or device in the normal way. For example, the following commands send information about **sub1.r** to the display and to the file **dispfile**:

```
cnm68 sub1.r
cnm68 sub1.r > dispfile
```

The first line listed by **cnm68** for an object module has the following format:

```
file (module): code: cc data: dd udata: uu total: tt (0xhh)
```

where

- *file* is the name of the file containing the module;
- *module* is the name of the module. If the module is unnamed, this field and its surrounding parentheses are not printed;
- *cc* is the number of bytes in the module code segment, in decimal;
- *dd* is the number of bytes in the modules' initialized data segment, in decimal;
- *uu* is the number of bytes in the module uninitialized data segment, in decimal;
- *tt* is the total number of bytes in the module's three segments, in decimal;
- *hh* is the total number of bytes in the module's three segments, in hexadecimal.

If **cnm68** displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the module code segments, initialized data segments, and uninitialized data segments, and the sum of the sizes of all segments of all modules. Each sum is in decimal; the total of all segments is also given in hexadecimal.

Options

cnm68 supports the following options:

-s	display size only
-l	display each symbol on a separate line
-o	prefix symbols with filename

Option **-s** tells **cnm68** to display only the sizes of the object modules. If this option is not specified, **cnm68** also displays information about each named symbol in the object modules.

Option **-l** tells **cnm68** to display the information in long form, by displaying each symbol's information on a separate line and by displaying a symbol's entire name. If this option is not used, **cnm68** displays the information about several symbols on a line and only displays the first eight characters of a symbol name.

Option **-o** tells **cnm68** to prefix each line generated for an object module with the name of the file containing the module and the module name in parentheses (if the module is named). If this option is not specified, this information is listed just once for each module, prefixed to the first line generated for the module.

Option **-o** is useful when using **cnm68** in combination with **grep**. For example, the following commands display all information about the module **perror** in the library **c.lib**:

```
cnm68 -o c.lib > tmp
grep perror tmp
```

Symbol Format

cnm68 displays information about a module's "named" symbols, e.g., about the symbols that begin with something other than a period followed by a digit. For example, the symbol **quad** is named, so information about it would be displayed; the symbol **.0123** is unnamed, so information about it would not be displayed.

For each named symbol in a module, **cnm68** displays its name, a two-character code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

Symbol Types

If the first character of a symbol type code is lowercase, the symbol can only be accessed by the module; in other words, the symbol is local to the module. If this character is uppercase, the symbol is global to the module: Either the module has defined the symbol and is allowing other modules to access it, or the module needs to access the symbol that must be defined as a global or public symbol in another module. The type codes are:

ab The symbol was defined using the assembler's **equ** directive. The value listed is the equated value of its symbol.

The compiler does not generate symbols of this type.

bs The symbol is in the uninitalized data segment. The value is the space reserved for the symbol.

The compiler generates **bs** symbols for static, uninitialized variables that are declared outside all functions and that are not dimensionless arrays.

The assembler generates **bs** symbols for symbols defined using the **bss** assembler directive.

dt The symbol is in the initialized data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables that are declared outside any function. Static variables are local to the program and so have type **dt**; all other variables are GLOBAL, i.e., accessible from other programs, and hence have type **Dt**.

Gl The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates **Gl** symbols for nonstatic, uninitialized variables that are declared outside all functions and that are not dimensionless arrays.

The assembler generates **Gl** symbols for variables declared using the **global** directive that have a nonzero size.

pg The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names. Static functions are local to the function, and so have type **pg**; all other functions are global and hence have type **Pg**.

un The symbol is used but not defined within the program. The value has no meaning.

In assembly language terms, a type of **Un** (the U is capitalized) indicates that the symbol is the operand of a **public** directive and that it is perhaps referenced in the operand field of some statements, but that the program did not create the symbol in a statement *label* field.

coff68 convert program to COFF format

SYNOPSIS

coff68 *infile* [*options*]

DESCRIPTION

coff68 converts the output of the **ln68** linker (both the executable program and source debugging information) to UNIX System 5 COFF format.

infile is the name of the file that contains the **ln68** generated executable program.

coff68 writes the COFF format information to a file whose name is derived by changing the extension of *infile* to **.cof**.

coff68 supports the following options:

-l Generate a listing file. The name of this file is created from *infile* by changing the extension to **.dls.**

-v Be verbose.

For example, the following commands convert the C source code in **exmpl.c** into COFF code in **exmpl.cof**:

```
c68 -n exmpl.c
ln68 -g exmpl.r -lc
coff68 exmpl
```

crc utility for generating the CRC for files

SYNOPSIS

crc *files*

DESCRIPTION

crc computes a number, called the CRC, for specified files. By using the standard "wild-card" characters, *files* can specify multiple files.

The CRC for a file is entirely dependent on the file's contents, and it is very unlikely that two files whose contents are different will have the same CRCs. Thus, **crc** can be used to determine whether a file has the expected contents.

The file **crclist** that is on the Aztec C disk lists the CRC values for each of the files on the disks. By comparing these values with those computed by your own running of **crc**, you can easily determine whether you have received all of the files to which you are entitled.

As an example of the usage of **crc**, the following command computes the **crc** of all files whose extension is **.c**:

```
crc *.c
```

diff compare source files

SYNOPSIS

diff [-b] *file1 file2*

DESCRIPTION

The **diff** utility is designed to display differences between two text files. **diff** will show all differences between *file1* and *file2*, along with information on how to make them similar. **diff** will display the exact lines that are different between the two files, including the relative line numbers in the files.

The Conversion List

diff writes a conversion list to its standard output that describes the changes that need to be made to *file1* to convert it into *file2*. The list is organized into a sequence of items, each of which describes one operation that must be performed on *file1*.

Conversion Items

There are three types of operations that can be specified in a conversion list item:

- adding lines to *file1* from *file2*
- deleting lines from *file1*
- replacing (changing) *file1* lines with *file2* lines.

A conversion list item consists of a command line, followed by the lines in the two files that are affected by the item's operation.

The Command Line

An item's command line contains a letter describing the operation to be performed: **a** for adding lines, **d** for deleting lines, and **c** for changing lines.

Preceding and following the letter are the numbers of the lines in *file1* and *file2*, respectively, that are affected by the command. If a range of lines in a file is affected, only the beginning and ending line numbers are listed, separated by a comma.

For example, the command line

```
5a3
```

says to add line 3 of **file2** after line 5 of **file1**. The command line

```
16a8,10
```

says to add lines 8, 9, and 10 of **file2** after line 16 of **file1**. The command line

```
100,150d75
```

says to delete lines 100 through 150 from **file1**, and that the last line in **file2** that matched a **file1** line was number 75.

The command

```
32c33
```

says to replace (change) line 32 in **file1** with line 33 in **file2**. The command

```
453,500c490,499
```

says to replace lines 453 through 500 in **file1** with lines 490 through 499 in **file2**.

The Affected Lines

As mentioned above, the lines affected by a conversion item's operation are listed after the item's command line. The affected lines from **file1** are listed first, flagged with a preceding < . Then come the affected lines from **file2**, flagged with a preceding > . The **file1** and **file2** lines are separated by the line

```
---
```

For example, the following conversion item says to add line 6 of **file2** after line 4 of **file1**. Line 6 of **file2** is `for (i=1; i<10;++i)`:

```
4a6
> for (i=1; i<10;++i)
```

Since no lines from **file1** are affected by an "add" conversion item, only the **file2** lines that will be added to **file1** are listed, and the separator line "---" is omitted.

The following conversion item says to delete lines 100 and 101 from **file1**, and that the last **file2** line that matched a **file1** line was numbered 110 . The deleted lines were `int a;` and `double b;`. Only the deleted lines are listed, and the separator line "---" is omitted:

```
100,101d110
< int a;
< double b;
```

The following conversion item says to replace lines **53** through **56** in **file1** with lines **60** and **61** in **file2**. Lines **53** through **56** in **file1** are **if (a=b){, d=a;, a++;,** and **}**. Lines **60** and **61** of **file2** are **if (a==b)** and **d=a++;**

```
53,58c60,61
< if (a=b){
< d = a;
< a++;
< }
---
> if (a==b)
> d = a++;
```

The -b Option

Option **-b** causes **diff** to ignore trailing blanks (spaces and tabs) and to consider strings of blanks to be identical. If this option is not specified, **diff** considers two lines to be the same only if they match exactly. For example, if **file1** contains the line

```
^abc$
```

(**^** and **$** stand for "the beginning of the line" and "the end of the line," respectively, and are not actually in the file) and if **file2** contains the line

```
^abc $
```

then **diff** would consider the two lines to be the same or different, depending on whether or not it was started with option **-b**.

And **diff** would consider the lines

```
^a          b c$
```

and

```
^a b c$
```

to be the same or different, depending on whether or not it was started with option **-b**.

diff will never consider blanks to match a null string, regardless of whether **-b** was used or not. So **diff** will never consider the lines

```
^abc$
```

and

```
^a bc$
```

to be the same.

Differences Between the UNIX and Manx Versions

The Manx and UNIX versions of **diff** are actually most similar when the latter program is invoked with option **-H**. As with the UNIX **diff** when used with option **-H**, the Manx **diff** works best when changed stretches are short and well separated, and works with files of unlimited length.

Unlike the UNIX **diff**, the Manx **diff** does not support option **-E**, **-F**, or **-H**. Unlike the UNIX **diff**, the Manx version requires that both operands be actual files. Because of this, the Manx version of **diff** does not support the features of the UNIX version that allow one operand to be a directory name, (to specify a file in that directory having the same name as the other operand), and that allow one operand to be '-' (to specify the **diff** standard input instead of a file).

grep pattern matching program

SYNOPSIS

grep [*options*] *pattern* [*file1 file2...*]

DESCRIPTION

grep is a program, similar to the UNIX **grep**, that searches a specified group of files for lines containing a specified pattern and writes the lines to its standard output.

For example, the following command examines all files in the current directory that have extension **.c** and prints those lines that contain the word **hello**.

```
grep hello *.c
```

Options

The following options are supported by **grep**:

- **-c** Print just the name of each file and the number of matching lines that it contained.
- **-f** A character in the pattern will match both its upper- and lowercase equivalent.
- **-l** Print only the names of the files that contain matching lines.
- **-n** Precede each matching line that is printed by its relative line number within the file that contains it.
- **-v** Print all lines that do not match the pattern.

Input Files

file1, file2,... specify the files that **grep** is to search. If no files are specified, **grep** searches its standard input. Each filename can specify a single file to be searched. A name can also specify a class of files to be searched, using the special characters * and ?. The character * matches any string of characters in a filename, and ? matches any single character. For example,

```
grep int main.c sub1.c sub2.c
```

searches **main.c**, **sub1.c**, and **sub2.c** for the string **int**. The command

```
grep int *.c
```

searches all files whose extension is **c** for the string **int**. The command

```
grep int a*.txt b*.doc
```

searches for the string **int** in each file (1) whose extension is **.txt** and first character is **a** and (2) whose extension is **.doc** and first character is **b**. The command

```
grep int sub?.c
```

searches for the string **int** in each file whose filename contains four characters, the first three being **sub**, and whose extension is **.c**.

Patterns

A pattern consists of a limited form of regular expression. It describes a set of character strings, any of whose members are said to be matched by the regular expression.

The patterns recognized by **grep** are:

x	an ordinary character (i.e. one other than the following special characters) matches itself.
.	matches anything except carriage return or new line
[*chars*]	character matches any one of the characters that are between the brackets. For example, **[ad9@]** matches any of the characters **a, d, 9,** or **@**.
[^*chars*]	matches any one character except those ones within the brackets. For example, **[^ad9@]** matches any character except **a, d, 9,** or **@**.
^	matches the beginning of the line. For example, **^main** matches a line that begins with the word **main**, but not one that begins with **the main**.
$	matches the end of the line.
x*	matches 0 or more occurrences of the character x

A backslash, \, followed by any of the above characters matches the specified character.

Examples

Example: Matching Any Character

Suppose you want to find all lines in the file **prog.c** that contain a four-character string whose first and last characters are **m** and **n**, respectively, and whose other characters you do not care about. The command

```
grep m..n prog.c
```

will do the trick, since the special character "." matches any single character.

Example: Matching Any of a Set of Characters

Suppose that you want to find all lines in the file **file.doc** that begin with a digit. The command

```
grep ^[0123456789] file.doc
```

will do just that. This command can be abbreviated as

```
grep ^[0-9] file.doc
```

And if you wanted to print all lines that do not begin with a digit, you could enter

```
grep ^[^0-9] file.doc
```

Example: Matching Line Beginning and End

Suppose you want to find the number of the line on which the definition of the function **add** occurs in the file **arith.c**. Entering

```
grep -n add arith.c
```

is not good, because it will print lines in which **add** is called, in addition to the line you are interested in. Assuming that you begin all function definitions at the beginning of a line, you could enter

```
grep -n ^add arith.c
```

to accomplish your purpose.

The character **$** is a companion to **^**, and stands for "the end of the line." So if you want to find all lines in **file.doc** that end in the string **time**,you could enter

```
grep time$ file.doc
```

And the following will find all lines that contain just **.PP**:

```
grep ^.PP$ file.doc
```

Example: Matching Repeated Characters

Suppose you want to find all lines in the file **prog.c** that contain strings whose first character is **e** and whose last character is **z**. The command

```
grep e.*z prog.c
```

will do that. The **e** matches an **e**, the ***** matches zero or more occurrences of the character that precedes it (in this case a "." which matches anything) and the **z** matches a **z**.

Example: Matching a Special Character

There are occasions when you want to find the character "." in a file, and do not want **grep** to consider it to be special. In this case, you can use the backslash character, \, to turn off the special meaning of the next character.

For example, suppose you want to find all lines containing

```
.PP
```

Entering

```
grep .PP prog.doc
```

is not adequate, because it will find lines such as

```
THE APPLICATION OF
```

since the "." matches the letter **A**. But if you enter

```
grep \.PP prog.doc
```

grep will print only what you want.

The backslash character can be used to turn off the special meaning of any special character. For example,

```
grep \\n prog.c
```

finds all lines in **prog.c** containing the string **\n**.

Example; Simple String Matching

The following command will search the files **file1.txt** and **file2.txt** and print the lines containing the word **heretofore**:

```
grep heretofore file1.txt file2.txt
```

If you are not interested in the specific lines of these files, but just want to know the names of the files containing the word **heretofore**, you could enter

```
grep -l heretofore file1.txt file2.txt
```

The above two examples ignore lines in which **heretofore** contains capital letters, such as when it begins a sentence. The following command will cover this situation:

```
grep -lf heretofore file1.txt file2.txt
```

grep processes all options at once, so multiple options must be specified in one dash parameter. For example, the command

```
grep -l -f heretofore file1.txt file2.txt
```

will not work.

Differences Between the Manx and UNIX Versions

The Manx and UNIX versions of **grep** differ in the options they accept and the patterns they match.

Option Differences

- Option **-f** is supported only by the Manx **grep**.
- Options **-b** and **-s** are supported only by the UNIX **grep**.

Pattern Differences

Basically, the patterns accepted by the Manx **grep** are a subset of those accepted by the UNIX **grep**.

- The Manx **grep** does not allow a regular expression to be surrounded by \(and \).
- The Manx **grep** does not accept the construct \{m\}.
- The Manx **grep** does not allow a right bracket,], to be specified within brackets.

hd hex dump utility

SYNOPSIS

hd[+n[.]] *file1* [+n[.]] *file 2* ...

DESCRIPTION

hd displays the contents of one or more files in hex and ASCII to its standard output.

file1, *file2*, ... are the names of the files to be displayed.

+n specifies the offset into the files where the display is to start and defaults to the beginning of the file. If **+n** is followed by a period, **n** is assumed to be a decimal number; otherwise, it is assumed to be hexadecimal. Each file will be displayed beginning at the last specified offset.

Examples

To display the data forks of the files **oldtest** and **newtest**, beginning at offset **0x16b** and of the file named **junk** beginning at its first byte, you would use the command:

```
hd +16b oldtest newtest +0 junk
```

To display the contents of **tstfil**, beginning at byte **1000**, you would use the command:

```
hd +1000. tstfil
```

hex68 Convert Program to Intel Hex-Code

SYNOPSIS

hex68 [*-options*] *prog*

DESCRIPTION

hex68 translates the program that is in the file named *prog*, and that was generated by the Aztec C68k/ROM linker, into Intel hex code. The program can then be burned into ROM by feeding the hex code into a ROM programmer. The hex code is written to one or more files, each of which contains the hex code for one ROM chip.

The ROM chips that are generated from the **hex68** output files will contain the program's code, followed by a copy of its initialized data.

Note: when a ROM system is started, its RAM contains random values; the Aztec C68k/ROM startup routine sets up its initialized data area, using the copy that is in ROM.

hex68 assumes that the size of each ROM chip is 2kb. You can explicitly define the size of each ROM using **hex68**'s **-p** option.

The Output Files

Even- and Odd-addressed Bytes in the Same Chips

hex68 can optionally generate hex code so that the program's even-addressed bytes are in one set of ROM chips, and its odd-addressed bytes are in another. We will discuss this option below. In this section we discuss the output files that are created when this option is not used; i.e. when a program's even- and odd-addressed bytes are in the same set of ROM chips.

When neither **-e** nor **-o** is specified, **hex68** derives the name of each output file from that of the input file, by appending an extension of the form **.h***nn*, where *nn* is a number. For example, if the name of the linker-generated file is **prog**, then the names of the output files generated by **hex68** are **prog.h00**, **prog.h01**, and so on, where the **.h00** file contains the hex code for the lowest-addressed ROM, **.h01** the hex code for the next ROM, etc.

For example, suppose that **hex68** is creating Intel hex code for a program whose code and copy of initialized data will reside in three 2kb ROMs that begin at location 0. Then **hex68** will create the following files:

prog.h00 Contains the Intel hex code for the ROM chip that occupies addresses **0-0x7ff;**

prog.h01 Contains the hex code for the ROM that occupies **0x800-0xfff;**

prog.h02 Contains the hex code for the ROM that occupies **0x1000-0x17ff.**

Even- and Odd-addressed Bytes in Separate Chips

To place a program's even-addressed bytes in one set of ROM chips and its odd-addressed bytes in another, you must run **hex68** twice: once using the **-e** option to generate the hex code for the chips that contain the even-addressed bytes, and once using the **-o** option to generate hex code for the chips that contain the odd-addressed bytes.

When either **-e** or **-o** is specified, **hex68** generates one or more files, each of which contains the Intel hex code for one ROM chip. By default, the size of each chip is 2kb, but you can use the **-p** option to explicitly define the chip size.

When the **-e** option is specified, the extension of the files are of the form **.e***nn*, where *nn* is a decimal number. The **.e00** file contains the hex code for the first of the ROM chips that contain even-addressed bytes, the **.e01** file contains the hex code for the second ROM chip, and so on.

When the **-o** option is specified, the extension of the files are of the form **.o***nn*, where *nn* is a decimal number. The **.o00** file contains the Intel hex code for the first of the ROM chips that contain odd-addressed bytes, the **.o01** file contains the hex code bytes for the second ROM chip, and so on.

The Options

hex68 supports the following options:

-b*x* The program begins *x* bytes into the first ROM chip, where *x* is a hexadecimal number. If this option isn't specified, the program begins at the beginning of the first ROM chip.

-e Output hex code for the program's even-addressed bytes.

-o Output hex code for the program's odd-addressed bytes.

-p*n* The size of each ROM is *n* kilobytes, where *n* is a decimal number. If this option isn't specified, the size defaults to 2kb. For example, the following command specifies that each ROM chip is 64kb long:

```
hex68 -p64 exmpl
```

lb68 object file librarian

SYNOPSIS

lb68 *library* [*options*][*mod1 mod2 ...*]

DESCRIPTION

lb68 is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.

Arguments and Options

The *library* Argument

lb68 acts upon a single library file. The first argument to **lb68** (*library*, in the synopsis) is the name of this file. The filename extension for library is optional; if not specified, it is assumed to be **.lib**.

The *options* Argument

There are function code options and qualifier options. These options are summarized and then described in detail.

Function Code Options

lb68 performs one function at a time on the specified library. The functions that **lb68** can perform, and the corresponding option codes, are:

Function	Code
create a library	(no code)
add modules to a library	**-a,-i, -b**
list library modules	**-t**
move modules in a library	**-m**
replace modules	**-r**
delete modules	**-d**
extract modules	**-x**
ensure module uniqueness	**-u**
define module extension	**-e**
read function from file	**-f**
help	**-h**

In the synopsis, the options argument is surrounded by square brackets. This indicates that the argument is optional; if a code is not specified, **lb68** assumes that a library is to be created.

Qualifier Options

In addition to a function code, the options argument can optionally specify a qualifier that modifies lb68 behavior as it is performing the requested function. The qualifiers and their codes are:

verbose	**-v**
silent	**-s**

The qualifier can be included in the same argument as the function code, or as a separate argument. For example, to cause lb68 to append modules to a library, and be silent when doing so, any of the following option arguments could be specified:

```
-as
-sa
-a -s
-s -a
```

The *mod* Argument

The arguments *mod1*, *mod2*, etc. are the names of the object modules, or the files containing these modules, that lb68 is to use. For some functions, lb68 requires an object module name, and for others it requires the name of a file containing an object module. In the latter case, the file's extension is optional; if not specified, lb68 will assume that it is **.o**. You can explicitly define the default module extension using option **-e**, described further in the section "Defining the Default Module Extension" at the end of lb68.

Reading Arguments from Another File

lb68 has a special argument, **-f** *filename*, that causes it to read command line arguments from the specified file. When done, it continues reading arguments from the command line. Arguments can be read from more than one file, but the file specified in a **-f** *filename* argument cannot itself contain a **-f** *filename* argument.

Basic Features

This section describes the basic features of lb68. The basic points you need to know about lb68 are:

- How to create a library
- How to list the names of modules in a library
- How modules get their names
- Order of modules in a library
- Getting lb68 arguments from a file.

How to Create a Library

A library is created by starting lb68 with a command line that specifies the name of the library file to be created and the names of the files whose object modules are to be copied into the library. It does not contain a function code, and it is this absence of a function code that tells lb68 that it is to create a library.

For example, the following command creates the library **exmpl.lib**, copying into it the object modules that are in the files **obj1.o** and **obj2.o**:

```
lb68 exmpl.lib obj1.o obj2.o
```

Making use of the lb68 assumptions about filenames for which no extension is specified, the following command is equivalent to the above command:

```
lb68 exmpl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by lb68. In this case, all the modules in the input library are copied into the new library.

When lb68 creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, lb68 erases the file having the same name as the specified library, and then renames the new library, giving it the name of the specified library. Thus, lb68 makes sure it can create a library before erasing an existing one.

Note that there must be room on the disk for both the old library and the new.

How to List Names of Modules in a Library

To list the names of the modules in a library, use the lb68 option **-t**. For example, the following command lists the modules that are in **exmpl.lib**:

```
lb68 exmpl.lib -t
```

The list includes some **DIR** entries. These identify blocks within the library that contain control information. They are created and deleted automatically as needed, and cannot be changed by you.

How Modules Get Their Names

When a module is copied into a library from a file containing a single object module (that is, from an object module generated by the Manx assembler), the name of the module within the library is derived from the name of the input file by deleting the input file's volume, path, and extension components.

For example, in the example given above, the names of the object modules in **exmpl.lib** are **obj1** and **obj2**.

An input file can itself be a library. In this case, a module's name in the new library is the same as its name in the input library.

Order of Modules in a Library

The order of modules in a library is important, since the linker makes only a single pass through a library when it is searching for modules. For a discussion of this, see the "Introduction to Linking" section of the **Linker** chapter.

When **lb68** creates a library, it places modules in the library in the order in which it reads them. Thus, in the example given above, the modules will be in the library in the following order:

```
obj1 obj2
```

As another example, suppose that the library `oldlib.lib` contains the following modules, in the order specified:

```
sub1 sub2 sub3
```

If the library `newlib.lib` is created with the command

```
lb68 newlib mod1 oldlib.lib mod2 mod3
```

the contents of the newly-created `newlib.lib` will be:

```
mod1 sub1 sub2 sub3 mod2 mod3
```

The **ord68** utility program can be used to create a library whose modules are optimally sorted. For information, see the **ord68** description later in this chapter.

Getting Arguments From a File

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to **lb68** on a single command line. In this case, the **lb68 -f** *filename* feature can be of use: when **lb68** finds this option, it opens the specified file and starts reading command arguments from it. After finishing the file, it continues to scan the command line.

For example, suppose the file `build` contains the line

```
exmpl obj1 obj2
```

Then entering the command

```
lb68 -f build
```

causes `lb68` to get its arguments from the file `build`, which causes `lb68` to create the library `exmpl.lib` containing `obj1` and `obj2`.

Arguments in a **-f** *filename* can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a **-f** *filename* can be on separate lines, if desired.

The **lb68** command line can contain multiple **-f** arguments, allowing **lb68** arguments to be read from several files. For example, if some of the object modules that are to be placed in **exmpl.lib** are defined in **arith.inc**, **input.inc**, and **output.inc**, then the following command could be used to create **exmpl.lib**:

```
lb68 exmpl -f arith.inc -f input.inc -f output.inc
```

A **-f** *filename* can contain any valid **lb68** argument, except for another **-f** . That is, **-f** files cannot be nested.

Advanced Features

In this section we describe the rest of the functions that **lb68** can perform. These primarily involve manipulating selected modules within a library.

Adding Modules to a Library

lb68 allows you to add modules to an existing library. The modules can be added before or after a specified module in the library or can be added to the beginning or end of the library.

The options that select the add operator are:

Option	Function
-b *target*	add modules before the module *target*
-i *target*	same as **-b** *target*
-a *target*	add modules after the module *target*
-b+	add modules to the beginning of the library
-i	same as **-b+**
-a+	add modules to the end of the library

In an **lb68** command that selects the add operator, the names of the files containing modules to be added follows the add option code (and the target module name, when appropriate). A file can contain a single module or a library of modules.

Modules are added in the order that they are specified. If a library is to be added, its modules are added in the order they occur in the input library.

Adding Modules Before an Existing Module

As an example of the addition of modules before a selected module, suppose that the library **exmpl.lib** contains the modules

```
obj1 obj2 obj3
```

The command

```
lb68 exmpl -i obj2 mod1 mod2
```

adds the modules in the files **mod1.r** and **mod2.r** to **exmpl.lib**, placing them before the module **obj2**. The resultant **exmpl.lib** looks like this:

```
obj1 mod1 mod2 obj2 obj3
```

As an example of the addition of one library to another, suppose that the library **mylib.lib** contains the modules

```
mod1 mod2 mod3
```

and that the library **exmpl.lib** contains

```
obj1 obj2 obj3
```

Then the command

```
lb68 -b obj mylib.lib
```

adds the modules in **mylib.lib** to **exmpl.lib**, resulting in **exmpl.lib** containing

```
obj1 mod1 mod2 mod3 obj2 obj3
```

Note that in this example, we had to specify the extension of the input file **mylib.lib**. If we had not included it, lb68 would have assumed that the file was named **mylib.o**.

Adding Modules After an Existing Module

As an example of adding modules after a specified module, the command

```
lb68 exmpl -a obj1 obj2 obj3
```

will insert **mod1** and **mod2** after **obj1** in the library **exmpl.lib**. If **exmpl.lib** originally contained

```
obj1 obj2 obj3
```

then after the addition, it contains

```
obj1 mod1 mod2 obj2 obj3
```

Adding Modules at the Beginning or End of a Library

Options **-b+** and **-a+** tell **lb68** to add the modules whose names follow the option to the beginning or end of a library, respectively. Unlike options **-i** and **-a**, these options are not followed by the name of an existing module in the library.

For example, given the library **exmpl.lib** containing

```
obj1 obj2
```

the following command will add the modules **mod1** and **mod2** to the beginning of **exmpl.lib**:

```
lb68 exmpl -i+ mod1 mod2
```

resulting in **exmpl.lib** containing

```
mod1 mod2 obj1 obj2
```

The following command will add the same modules to the end of the library:

```
lb68 exmpl -a+ mod1 mod2
```

resulting in **exmpl.lib** containing

```
obj1 obj2 mod1 mod2
```

Moving Modules in a Library

Modules which already exist in a library can be easily moved about, using the move option, **-m**.

As with the options for adding modules to an existing library, there are several forms of move functions:

Option	Meaning
-mb *target*	move modules before the module *target*
-ma *target*	move modules after the module *target*
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the **lb68** command, the names of the modules to be moved follow the move option code.

The modules are moved in the order in which they are found in the original library, not in the order in which they are listed in the **lb68** command.

Deleting Modules

Modules can be deleted from a library using option **-d**. The command for deletion has the form

lb68 *libname* **-d** *mod1 mod2 ...*

where *mod1*, *mod2*, ... are the names of the modules to be deleted.

For example, suppose that **exmpl.lib** contains

```
obj1 obj2 obj3 obj4 obj5 obj6fP
```

The following command deletes **obj3** and **obj5** from this library:

```
lb68 exmpl -d obj3 obj5
```

Replacing Modules

The replace option is used to replace one module in a library with one or more other modules.

The replace option has the form **-r** *target*, where *target* is the name of the module being replaced. In a command that uses the replace option, the names of the files whose modules are to replace the target module follow the replace option and its associated target module. Such a file can contain a single module or a library of modules.

Thus, an lb68 command to replace a module has the form:

lb68 *library* **-r** *target mod1 mod2 ...*

For example, suppose that the library **exmpl.lib** looks like this:

```
obj1 obj2 obj3 obj4
```

Then to replace **obj3** with the modules in the files **mod1.o** and **mod2.o**, the following command could be used:

```
lb68 exmpl -r obj3 mod1 mod2
```

resulting in **exmpl.lib** containing

```
obj1 obj2 mod1 mod2 obj4
```

Uniqueness

lb68 allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

Option **-u** causes lb68 to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, option **-u** causes lb68 to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, if the library **expl.lib** contains the following:

```
obj1 obj2 obj3 obj1 obj3
```

The command

```
lb68 expl -u
```

will delete the second copies of the modules **obj1** and **obj2**, leaving the library looking like this:

```
obj1 obj2 obj3
```

Extracting Modules

Extracting modules from a library option **-x** extracts modules from a library and puts them in separate files, without modifying the library.

The names of the modules to be extracted follow the **-x** option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it is written to a new file; the file has the same name as the module and extension **.o**.

For example, given the library **expl.lib** containing the modules

```
obj1 obj2 obj3
```

the command

```
lb68 expl -x
```

extracts all modules from the library, writing **obj1** to **obj1.o**, **obj2** to **obj2.o**, and **obj3** to **obj3.o**.

And the command

```
lb68 expl -x obj2
```

extracts just **obj2** from the library.

The Verbose Option

The verbose option, **-v**, causes **lb68** to be verbose; that is, to tell you what it is doing.

Silence Option

The silence option, **-s**, tells **lb68** not to display its sign on message.This option is especially useful when redirecting the output of a list command to a disk file.

Rebuilding a Library

The following commands provide a convenient way to rebuild a library:

```
lb68 exmpl -st > tfil
lb68 exmpl -f tfil
```

The first command writes the names of the modules in **exmpl.lib** to the file **tfil**. The second command then rebuilds the library, using as arguments the listing generated by the first command.

The **-s** option to the first command prevents **lb68** from sending information to **tfil** that would foul up the second command. The names sent to **tfil** include entries for the directory blocks, ****DIR****, but these are ignored by lb68.

Defining the Default Module Extension

Specification of the extension of an object module file is optional; lb68 assumes that the extension is **.o**. You can explicitly define the default extension using option **-e**. This option has the form:

-e .ext

For example, the following command creates a library; the extension of the input object module files is **.i**.

```
lb68 my.lib -e .i mod1 mod2 m
```

Help

Option **-h** will generate a summary of lb68 functions and options.

make program maintenance utility

SYNOPSIS

make [*options*] [*name1 name2 ...*]

DESCRIPTION

make is a program, similar to the UNIX program of the same name, whose primary function is to create, and keep up-to-date, files that are created from other files, such as programs, libraries, and archives.

When told to make a file, **make** first ensures that the files from which the target file is created are up-to-date or current, recreating only the ones that are not. Then, if the target file is not current, **make** creates it.

Interfile dependencies and the commands which must be executed to create files are specified in a file called the MAKEFILE, which you must write.

make has a rule-processing capability, which allows it to infer, without being explicitly told, the files on which a file depends and the commands which must be executed to create a file. Some rules are built into **make**; you can define others within the makefile.

A rule tells **make** something like this:

> "a target file having extension **.x** depends on the file having the same basic name and extension **.y**. To create such a target file, apply the commands ...."

Rules simplify the task of writing a makefile: A file's dependency information and command sequences need to be explicitly specified in a makefile only if this information cannot be inferred by the application of a rule.

make has a macro capability. A character string can be associated with a macro name; when the macro name is invoked in the makefile, it is replaced by its string.

Preview

The rest of this description of **make** is divided into the following sections:

1. The basics
2. Advanced features
3. Examples

The Basics

This section presents the basic features of **make**, with which you will be able to start using **make**. The second part of this chapter describes advanced features of **make**.

Before you can begin using **make**, you must know what it does, how to create a simple makefile that contains dependency entries, how to take advantage of **make**'s rule-processing capability, and, finally, how to tell **make** to create a file. Each of these topics is discussed in the following paragraphs.

What make Does

The main function of **make** is to make a target file "current," where a file is considered "current" if the files on which it depends are current and if it was modified more recently than its prerequisite files. To make a file current, **make** makes the prerequisite files current; then, if the target file is not current, **make** executes the commands associated with the file, which usually recreates the file.

As you can see, **make** is inherently recursive: Making a file current involves making each of its prerequisite files current, making these files current involves making each of their prerequisite files current, and so on.

make is very efficient: it only creates or recreates files that are not current. If a file on which a target file depends is current, **make** leaves it alone. If the target file itself is current, **make** will announce the fact and halt without modifying the target.

It is important to have the time and date set for **make** to behave properly, since it uses the last modified times that are recorded in the files directory entries to decide if a target file is not current.

The Makefile

When make starts, it first reads a file, which you must create, called the MAKEFILE. By default, **make** assumes this file is named **makefile**, but this can be overridden using **make**'s **-f** option. This file contains dependency entries defining interfile dependencies and the commands that must be executed to make a file current. It also contains rule definitions and macro definitions.

In the following paragraphs, we describe only dependency entries. In the "Advanced Features" section of this chapter we discuss the somewhat more advanced topics of rule and macro definition.

A dependency entry in a makefile defines one or more target files, the files on which the targets depend, and the operating system commands that are to be executed when any of the targets is not current. The first line of the entry specifies the target files and the files on which they depend; the line begins with the target filenames, followed by a colon, followed by one or more spaces or tabs, followed by the names of the prerequisite files.

Please Note:. It is important to place spaces or tabs after the colon that separates target and dependent files; on systems that allow colons in filenames, this allows **make** to distinguish between the two uses of the colon character.

The commands are on the following lines of the dependency information entry. The first character of a command line must be a tab or a space; **make** assumes that the command lines end with the last line not beginning with a tab or space.

For example, consider the following dependency entry:

```
prog: prog.r sub1.r sub2.r
    ln68 -o prog prog.r sub1.r sub2.r -lc
```

This entry says that the file **prog** depends on the files **prog.r, sub1.r, and sub2.r**. It also says that if **prog** is not current, **make** should execute the **ln68** command. **make** considers **prog** to be current if it exists and if it has been modified more recently than **prog.r, sub1.r,** and **sub2.r**.

The above entry describes only the dependence of **prog** on **prog.r, sub1.r,** and **sub2.r**. It does not define the files on which the **.r** files depend. For that, we need either additional dependency entries in the makefile or a rule that can be applied to create **.r** files from **.c** files.

For now, we will add dependency entries in the makefile for **prog.r, sub1.r,** and **sub2.r,** which will define the files on which the object modules depend and the commands to be executed when an object module is not current. Later we will add a rule to the makefile that will tell **make** how to create a **.r** file from a **.c** file.

Suppose that the **.r** files are created from the C source files **prog.c, sub1.c,** and **sub2.c;** that **sub1.c** and **sub2.c** contain a statement to include the file **defs.h;** and that **prog.c** does not contain any **#include** statements. Then the following makefile could be used to explicitly define all the information needed to create **prog**

```
prog: prog.r sub1.r sub2.r
    ln68 -o prog prog.r sub1.r sub2.r -lc
prog.r: prog.c
    c68 prog.c
sub1.r: sub1.c defs.h
    c68 sub1.c
sub2.r: sub2.c defs.h
    c68 sub2.c
```

This makefile contains four dependency entries: for **prog, prog.r, sub1.r,** and **sub2.r**. Each entry defines the files on which its target file depends and the commands to be executed when its target is not current. The order of the dependency entries in the makefile is not important.

We can use this makefile to make any of the four target files defined in it. If none of the target files exists, and if the name of the makefile is **makefile**, then entering

```
make prog
```

causes **make** to compile and assemble all three object modules from their C source files, and then create **prog** by linking the object modules together.

Suppose that you create **prog** and then modify **sub1.c**. Then telling **make** to make **prog** causes **make** to compile and assemble **sub1.c** only, and then recreate **prog**.

If you then modify **defs.h,** and tell **make** to create **prog, make** will compile and assemble **sub1.c** and **sub2.c,** and then recreate **prog**.

You can tell **make** to make any file defined as a target in a dependency entry. Thus, if you want to make **sub2.r** current, you could enter

```
make sub2.r
```

A makefile can contain dependency entries for unrelated files. For example, the following dependency entries can be added to the above makefile:

```
hello: hello.r
    ln68 hello.r -lc
hello.r: hello.c
    c68 hello.c
```

With these dependency entries, you can tell **make** to make **hello** and **hello.r**, in addition to **prog** and its object files.

Advanced Features

The last section presented the basic features of **make** to help you begin using **make**. This section presents the rest of **make**'s features.

Dependent Files

The target and source files that are in a dependent entry can be in different drives or directories with the following caveats:

- If the filename contains a colon (for example, because the filename defines the volume on which the file is located), the colon must be followed by characters other than spaces or tabs, so that **make** can distinguish between this use of the colon character and its use as a separator between the target and dependent files in a dependency line. This should not be a problem, since most systems do not allow filenames to contain spaces or tabs.

- All references to a file must use the same name. For example, if a file is referred to in one place using the name

  ```
  \root\src\foo.c
  ```

  then all references to the file must use this exact same name. The name

  ```
  ..\src\foo.c
  ```

  would not match.

Macros

make has a simple macro capability that allows character strings to be associated with a macro name and to be represented in the makefile by the name. The following paragraphs describe how to use macros within a makefile, then how they are defined, and finally some special features of macros.

Using Macros

Within a makefile, a macro is invoked by preceding its name with a dollar sign; macro names longer than one character must be parenthesized. For example, the following are valid macro invocations:

```
$(CFLAGS)
$2
$(X)
$X
```

The last two invocations are identical.

When **make** encounters a macro invocation in a dependency line or command line of a makefile, it replaces it with the character string associated with the macro. For example, suppose that the macro **OBJECTS** is associated with the string **a.r b.r c.r d.r**. Then the dependency entries:

```
prog: prog.r a.r b.r c.r d.r
    ln68 prog.r a.r b.r c.r d.r
a.r b.r c.r d.r: defs.h
```

within a makefile could be abbreviated as:

```
prog: prog.r $(OBJECTS)
    ln68 prog.r $(OBJECTS)
$(OBJECTS): defs.h
```

Defining Macros in a makefile

A macro is defined in a makefile by a line consisting of the macro name, followed by the character =, followed by the character string to be associated with the macro.

For example, the macro **OBJECTS**, used above, could be defined in the makefile by the line

```
OBJECTS = a.r b.r c.r d.r
```

A makefile can contain any number of macro definition entries. A macro definition must appear in the makefile before the lines in which it is used.

Defining Macros in a Command Line

A macro can be defined in the command line that starts **make**. The syntax for a command line definition has the following form:

make *MACRO=text*

For example, the following command assigns the value **-DFLOAT** to the macro **CFLAGS**:

```
make CFLAGS=-DFLOAT
```

Another example would be as follows:

```
make MACRO=text
```

Note that the equal sign (=) is the key to having it be a macro definition. If the macro is to be empty, enter:

```
make MACRO=
```

Command line macro definition always overrides makefile macro definition. If a macro has any whitespace, the macro name must be enclosed in quotes.

Macros Used by Built-In Rules

make has two macros, CFLAGS and AFLAGS, that are used by the built-in rules. These macros by default are assigned the null string. This can be overridden by a macro definition entry in the makefile.

For example, the following would cause **CFLAGS** to be assigned the string **-T**:

```
CFLAGS = -T
```

These macros are discussed below in the description of built-in rules.

Special Macros

There are three special macros: **$$**, **$***, and **$@**. **$$** represents the dollar sign. The other two are discussed below.

Before issuing any command, two special macros are set: **$@** is assigned the full name of the target file to be made, and **$*** is the name of the target file, without its extension. Unlike other macros, these can only be used in command lines, not in dependency lines.

For example, suppose that the files **x.c, y.c,** and **z.c** need to be compiled using the option **-DFLOAT**. The following dependency entry could be used:

```
x.r y.r z.r:
    c68 -DFLOAT $*.c
```

When **make** decides that **x.r** needs to be recreated from **x.c**, it will assign **$*** to the string **x**, and the command

```
c68 -DFLOAT x.c
```

will be executed. Similarly, when **y.r** or **z.r** is made, the command

```
c68 -DFLOAT y.c
```

or

```
c68 -DFLOAT z.c
```

will be executed.

The special macros can also be used in command lines associated with rules. In fact, the **$@** macro is primarily used by rules. We will discuss this more in the "Rules" section that follows.

Rules

The makefile describing a program built from many **.r** files would be huge if it had to explicitly state that each **.r** file depends on its **.c** source file and is made current by compiling its source file.

This is where rules are useful. When a rule can be applied to a file that **make** has been told to make or that is a direct or indirect prerequisite of it, the rule allows **make** to infer, without being explicitly told, the name of a file on which the target file depends and/or the commands that must be executed to make it current. This in turn allows makefiles to be very compact, only specifying information that **make** cannot infer by the application of a rule.

Some rules are built into **make**; you can define others in a makefile. In the rest of this section, we describe the properties of rules and **make**'s built-in rule for creating a **.o** file from a **.c** file.

make's Use of Rules

A rule specifies a target extension, source extension, and sequence of commands. Given a file that **make** wants to make, it searches the rules known to it for one that meets the following conditions:

- The rule's target extension is the same as the file's extension;
- A file exists that has the same basic name as the file **make** is working on and that has the rule's source extension.

If a rule is found that meets these conditions, **make** applies the first such rule to the file it is working on, as follows:

- The file having the source extension is defined to be a prerequisite of the file with the target extension;

- If the file having the target extension does not have a command sequence associated with it, the rule's commands are defined to be the ones that will make the file current.

Rule Definition

A rule consists of a source extension, target extension, and command list. In a makefile, an entry defining a rule consists of a line defining the two extensions, followed by lines containing the commands.

The line defining the extensions consists of the source extension, immediately followed by the target extension, followed by a colon.

All command lines associated with a rule must begin with a tab or space character. The first line following the extension line that does not begin with a tab or space terminates the commands for the rule.

For example, the following rule defines how to create a file having extension **.rel** from one having extension **.c**:

```
.c.rel:
    c68 -o $@ $*.c
```

The first line declares that the rule's source and target extension are **.c** and **.rel**, respectively.

The second line, which must begin with a tab, is the command to be executed when a **.rel** file is to be created using the rule.

Note the existence of the special macros **$@** and **$*** in the command line. Before the command is executed to create a **.rel** target file using the rule, the macro **$@** is replaced by the full name of the target file, and the macro **$*** by the name of the target, less its extension.

Thus, if **make** decides that the file **x.rel** needs to be created using this rule, it will issue the command

```
c68 -o x.rel x.c
```

If a rule defined in a makefile has the same source and target extensions as a built-in rule, the commands associated with the makefile version of the rule replace those of the built-in version.

Example

As an example of rules, we can simplify the sample makefile that is in "The Basics" section by adding a .c to .r rule. The makefile then becomes

```
.c.r:
    c68-o $@ $*.c
prog: prog.r sub1.r sub2.r
    ln68 -o prog prog.r sub1.r sub2.r
```

Interaction of Rules and Dependency Entries

As we showed in the above example, a rule allows you to leave some dependency information unspecified in a makefile. The **prog.r** entry in the long-winded makefile shown earlier in this section was not needed, because its information could be inferred by the **.o** to **.r** rule. And the dependence of **sub1.r** and **sub2.r** on their respective C source files, and the commands needed to create the object files was also not needed, since the information could be inferred from the **.o** to **.r** rule.

There are occasions when you do not want a rule to be applied; in this case, information specified in a dependency entry will override that which would be inferred from a rule. For example, the following dependency entry in a makefile

```
add.o:
    c68 -DFLOAT add.c
```

causes **add.o** to be compiled using the specified command rather than the command specified by the **.o** to **.r** rule. **make** still infers the dependence of **add.o** on **add.c**, using the **.o** to **.r** rule, however.

Built-in Rules

The following rules are built into **make**. The order of the rules is important, since **make** searches the list beginning with the first one, and applies the first applicable rule that it finds.

```
.c.o:
    cc $(CFLAGS) -o $@ $*.c
.asm.o:
    as $(AFLAGS) -o $@ $*.asm.
```

Thus, if both a **.asm** and a **.c** file exist with the same basic name, the **.c** file would be used and not the **.asm** file.

The two macros CFLAGS and AFLAGS that are used in the built-in rules are built into **make**, having the null character string as their values. To have **make** use other options when applying one of the built-in rules, you can define the macro in the makefile.

For example, if you want the options **-t** and **-DDEBUG** to be used when **make** applies the **.o** to **.r** rule, you can include the line

```
CFLAGS = -T -DDEBUG
```

in the makefile. Another way to accomplish the same result is to redefine the **.o** to **.r** rule in the makefile; this, however, would use more lines in the makefile than the macro redefinition.

For doing cross development with Aztec C68k/ROM, you will not be able to use **make**'s built-in rules, since these rules invoke the **cc** and **as** instead of **c68** and **as68**.

Commands

In this section we want to discuss the execution of operating system commands by **make**.

Allowed Commands

A command line in a dependency entry or rule within a makefile can specify any command that you can enter at the keyboard.

This includes batch commands, commands built into the operating system, and commands that cause a program to be loaded and executed from a disk file.

Logging Commands and Aborting make

Normally, before make executes a command, it writes the command to its standard output device; and when the command terminates, **make** halts if the command's return code was non-zero. Either or both of these actions can be suppressed for a command, by preceding the command in the makefile with a special character:

@	Tells **make** not to log the command;
-	Tells **make** to ignore the command's return code.

For example, consider the following dependency entry in a makefile:

```
prog: a.r b.r c.r d.r
    ln68 -o prog a.r b.r c.r d.r -lc
    @echo 'All done!'
```

when the **echo** command is executed, the command itself will not be logged to the console.

Long Command Lines

Makefile commands that start a Manx program, such as **c68**, **as68**, or **ln68**, can specify a command line containing up to 2048 characters.

For example, if a program depends on fifty modules, you could associate them with the macro **OBJECTS** in the makefile, and also include the dependency entry

```
prog: $(OBJECTS)
    ln68 -o prog $(OBJECTS) -lc
```

This will result in a very long command line being passed to **ln68**.

For the execution of other commands, the command line can contain at most 127 characters.

Makefile Syntax

This section has already presented most of the syntax of a makefile; that is, how to define rules, macros, and dependencies. However, we must discuss two features of a makefile syntax not presented elsewhere: comments and line continuation.

Comments

make assumes that any line in a makefile whose first character is # is a comment, and ignores it. For example:

```
#
# the following rule generates
# 68k object module
# from a C source file:
#
.c.r:
    c68 -pa $*.c
```

Line Continuation

Many of the items in a makefile must be on a single line: A macro definition, the file dependency information in a dependency entry, and a command that **make** is to execute must each be on a single line.

You can tell **make** that several makefile lines should be considered to be a single line by terminating each of the lines, except the last, with the backslash character, '\'. When **make** sees this, it replaces the current line's backslash and newline, and the next line's leading blanks and tabs by a single blank, thus effectively joining the lines together.

The maximum length of a makefile line after joining continued lines is 2048 characters.

For example, the following macro definition equates **OBJ** to a string consisting of all the specified object module names.

```
OBJ = printf.r fprintf.r format.r\
    scanf.r fscanf.r scan.r\
    getchar.r getc.r
```

As another example, the following dependency entry defines the dependence of **driver.lib** on several object modules, and specifies the command for making **driver.lib**:

```
driver.lib: driver.r printer.r \
          in.r \
          out.r
    lb68 -o driver.lib driver.r
          printer.r \
          in.r out.r
```

This second example could have been more cleanly expressed using a macro:

```
DRIVOBJ= driver.r printer.r\
          in.r out.r
driver.lib: $(DRIVOBJ)
    lib -o driver.lib $(DRIVOBJ)
```

This was done to show that dependency lines and command lines can be continued, too.

Starting make

We have already discussed how **make** is told to make a single file. Entering

make *filename*

makes the file named *filename*, which must be described by a dependency entry in the makefile. And entering

make

makes the first file listed as a target file in the first dependency entry in the makefile.

In both of these cases, **make** assumes the makefile is named *makefile* and that it is in the current directory on the default drive.

In this section we want to describe the other features available when starting **make**.

The Command Line

The complete syntax of the command line that starts **make** is:

make [-n] [-f *makefile*] **[-a] [***MACRO=str***] [***file1***] [***file2***] ...**

Square brackets indicate that the enclosed parameter is optional.

The parameters *file1*, *file2* ... are the names of the files to be made. Each file must be described in a dependency entry in the makefile. They are made in the order listed on the command line.

The other command line parameters are options and can be entered in upper- or lowercase. Their meanings are:

-n	Suppresses command execution. **make** logs the commands it would execute to its standard output device, but does not execute them.
-f *makefile*	The name of the makefile is *makefile* .
-a	Forces **make** to make all files upon which the specified target files directly or indirectly depend, and to make the target files, even those that it considers current.
MACRO=str	Creates a macro named *MACRO*, and assigns *str* as its value.

make's Standard Output

make only uses its standard output device for printing error messages and for logging commands. You can redirect **make**'s standard output device in the normal way.

The standard input and output devices of a program started by **make** inherit the standard input and output of the **make** program, unless the command that started the program explicitly redirected one or both of them.

Differences between the Manx and UNIX make Programs

The Manx **make** supports a subset of the features of the UNIX **make**. The following comments present features of the UNIX **make** that are not supported by the Manx **make**.

- The UNIX **make** will let you make a file that is not defined as a target in a makefile dependency entry, so long as a rule can be applied to create it. The Manx **make** does not allow this. For example, if you want to create the file **hello.o** from the file **hello.c** you could say, on UNIX

```
make hello.o
```

  even if **hello.o** was not defined to be a target in a makefile dependency entry. With the Manx **make**, you would have to have a dependency entry in a makefile that defines **hello.o** as a target.

- The UNIX **make** supports the following options, which are not supported by the Manx **make**:

  p, i, k, s, r, b, e, m, t, d, q

- The Manx **make** supports the option **-a**, which is not supported by the UNIX **make**.

- The special names .DEFAULT, .PRECIOUS, .SILENT, and .IGNORE are supported only by the UNIX **make**.

- Only the UNIX **make** allows the makefile to be read from **make**'s standard input.

- Only the UNIX **make** supports the special macros **$?** and **$%** and allows an uppercase D or F to be appended to the special macros, which thus modifies the meaning of the macro.

- Only the UNIX **make** requires that the suffixes for additional rules be defined in a .SUFFIXES statement.

- Only the UNIX **make** allows a target to depend on a member of a library or archive.

Examples

Example 1

This example shows a makefile for making several programs. Note the entry for **all**. This does not result in the generation of a file called **all**; it is only used so that **prog1** and **prog2** can be generated by typing **make** or **make all**.

```
#
# rules:
#
.c.r:
    c68 -o $@ $*.c
all: prog1 prog2
#
# macros:
#
OBJ=make.r parse.r scandir.r \
dumptree.r rules.r command.r
#
# dependency entry for making prog1:
prog1: $(OBJ)
    ln68 -o prog1 $(OBJ) -lc
    $(OBJ): parse.h lex.h
#dependency entry for making prog2
#prog2: prog2.r
    ln68 -o prog2
          prog2.r -lc
```

Example 2

The next example uses **make** to make a library, **my.lib**. Three directories are involved: the directory **libc** and two of its subdirectories, **sys** and **misc**. The C and assembly language source files are in the two subdirectories. There are makefiles named **makefile** in each of the three directories, and this example makes use of all of them. With the current directory being **libc**, you enter

```
make my.lib
```

This starts **make**, which reads the makefile in the **libc** directory. **make** will change the current directory to **sys** and then start another **make** program.

This second **make** compiles and assembles all the source files in the **sys** directory, using the makefile that is in the **sys** directory.

When the **sys make** finishes, the **libc make** regains control, and then starts yet another **make**, which compiles and assembles all the source files in the **misc** subdirectory, using the makefile that is in the **misc** directory.

When the **misc make** is done, the **libc make** regains control and builds **my.lib**. You can then remove the object files in the subdirectories by entering

```
make clean
```

The following files contain the makefiles for this example:

The makefile in the 'libc' Directory

```
my.lib: sys.mk misc.mk
    my.lib
    lb68 -o my.lib -f my.bld


sys.mk:
    cd sys
    make
    cd ..
misc.mk:
    cd misc
    make
    cd ..
clean:
    cd sys
    make clean
    cd ..
    cd misc
    make clean
    cd ..
```

The makefile for the 'sys' Directory

```
REL=asctime.r bdos.r begin.r chmod.r \
    croot.r csread.r ctime.r

COPT=
HEADER=../header
.c.r:
    c68 $(COPT) -I$(HEADER) $*.c -o $@
.asm.r:
    as68 $*.asm -o$@
all: $(REL)
clean:
```

The makefile for the 'misc' Directory

```
REL= atoi.r atol.r calloc.r ctype.r format.r \
    malloc.r \
    qsort.r sprintf.r sscanf.r fformat.r fscan.r
COPT=
HEADER=../header
.c.r:
    c68 $(COPT) -I$(HEADER) $*.c -o $@
.asm.r:
    as68 $*.asm -o $@
all: $(REL)
fformat.r: format.c
    c68 -I$(HEADER) -DFLOAT format.c -o fformat.r
fscn.o: scan.c
    c68 -I$(HEADER) -DFLOAT scan.c -o fscan.r
clean:
```

mkarcv create source archive

SYNOPSIS

mkarcv *arcfile*

DESCRIPTION

mkarcv combines several individual text files into one "archive" file, where the parameter *arcfile* is the name of the archive file. **mkarcv** reads the names of the files to be archived from its standard input device. Each filename is on a separate line.

Usually, a separate file containing the names of the files to be archived is created first and **mkarcv**'s standard input is redirected to this file in the standard manner. Alternatively, you can type in each file's name from the console, separating each name with a carriage return and ending the list with a period as the first character on a line by itself.

Files archived with **mkarcv** may be dearchived with the **arcv** utility described at the beginning of this chapter. Please note that files archived with **mkarcv** are not in any way altered or "squeezed," and the archive can be edited with a standard text editor.

EXAMPLE

To archive the files **explore.c** and **makefile** into the file **example.arc**, you would create a file named **input.txt** containing these two filenames with a carriage return after each name:

```
explore.c <CR>
makefile <CR>
```

and then redirect the standard input of **mkarcv** to this file:

```
mkarcv <input.txt example.arc
```

You can also interactively enter the filenames when you run the **mkarcv** command:

```
mkarcv example.arc <CR>
explore.c <CR>
makefile <CR>
. <cr>
```

obd68 list object code

SYNOPSIS

obd68 *objfile1* [, *objfile2,...*]

DESCRIPTION

obd68 lists the loader items that are in the Aztec object modules *objfile1*, *objfile2*, ...

ord68 sort object module list

SYNOPSIS

ord68 [-v] [*infile* [*outfile*]]

DESCRIPTION

ord68 sorts a list of object filenames for use by the **lb68** utility. A library that is generated from this sorted list will contain a limited number of backward references, i.e., global symbols that are defined in one module and referenced in a later module.

Since the specification of a library to the linker causes it to search the library only once, a library having no backward references must be specified only once when linking a program, and a library having backward references may need to be specified multiple times.

infile is the name of a file containing an unordered list of filenames. These files contain the object modules that are to be put into a library. If *infile* is not specified, this list is read from the **ord68** standard input. The filenames can be separated by space, tab, or newline characters.

outfile is the name of the file to which the sorted list is written. If it is not specified, the list is written to the **ord68** standard output. *outfile* can only be specified if *infile* is also specified.

Option **-v** causes **ord68** to be verbose, sending messages to its standard error device as it proceeds.

srec68 Convert program to Motorola S-records

SYNOPSIS

srec68 [*-options*] *prog*

DESCRIPTION

srec68 translates the program that's in the file named *prog*, and that was generated by the Aztec C68k/ROM linker, into Motorola S-records. The program can then be burned into ROM by feeding the S-records into a ROM programmer. The S-records are written to one or more files, each of which contains the hex code for one ROM chip. The ROM chips that are generated from the **srec68** output files will contain the program's code, followed by a copy of its initialized data.

Note: when a ROM system is started, its RAM contains random values; the Aztec C68k/ROM startup routine sets up its initialized data area, using the copy that is in ROM.

srec68 assumes that the size of each ROM chip is 2kb. You can explicitly define the size of each ROM using **srec68**'s **-p** option.

The Output Files

Even- and Odd-addressed Bytes in the Same Chips

srec68 can optionally generate S-records so that the program's even-addressed bytes are in one set of ROM chips, and its odd-addressed bytes are in another. This option is discussed below. In this section we discuss the output files that are created when this option isn't used; i.e. when a program's even- and odd-addressed bytes are in the same set of ROM chips.

When neither **-e** nor **-o** is specified, **srec68** derives the name of each output file from that of the input file, by appending an extension of the form **.m***nn*, where *nn* is a number. For example, if the name of the linker-generated file is **prog**, then the names of the output files generated by **srec68** are **prog.m00**, **prog.m01**, and so on, where the **.m00** file contains the S-records for the lowest-addressed ROM, **.m01** the S-records for the next ROM, etc.

For example, suppose that **srec68** is creating S-records for a program whose code and copy of initialized data will reside in three 2kb ROMs that begin at location **0**. Then **srec68** will create the following files:

prog.m00 Contains the S-records for the ROM chip that occupies addresses **0-0x7ff**;

prog.m01 Contains the S-records for the ROM that occupies **0x800-0xfff**;

prog.m02 Contains the S-records for the ROM that occupies **0x1000-0x17ff**.

Even- and Odd-addressed Bytes in Separate Chips.

To place a program's even-addressed bytes in one set of ROM chips and its odd-addressed bytes in another, you must run **srec68** twice: once using the **-e** option to generate the S-records for the chips that contain the even-addressed bytes, and once using the **-o** option to generate S-records for the chips that contain the odd-addressed bytes.

When either **-e** or **-o** is specified, **srec68** generates one or more files, each of which contains the S-records for one ROM chip. By default, the size of each chip is 2kb, but you can use the **-p** option to explicitly define the chip size.

When the **-e** option is specified, the extension of the files are of the form **.e***nn*, where *nn* is a decimal number. The **.e00** file contains the S-records for the first of the ROM chips that contain even-addressed bytes, the **.e01** file contains the S-records for the second ROM chip, and so on.

When the **-o** option is specified,the extension of the files are of the form **.o***nn*, where *nn* is a decimal number. The **.o00** file contains the S-records for the first of the ROM chips that contain odd-addressed bytes, the **.o01** file contains the S-records for the second ROM chip, and so on.

The Options

srec68 supports the following options:

-a*n* The size of an S-record's address field is *n* bytes, where (following Motorola specifications)*n* can be 2, 3, or 4. If this option isn't specified, the field size defaults to 2 bytes.

-b*x* The program begins *x* bytes into the first ROM chip, where *x* is a hexadecimal number. If this option isn't specified, the program begins at the beginning of the first ROM chip.

-e Output S-records for the program's even-addressed bytes.

-o Output S-records for the program's odd-addressed bytes.

-p*n* The size of each ROM is *n* kilobytes, where *n* is a decimal number. If this option isn't specified, the size defaults to 2kb. For example, the following command specifies that each ROM chip is 64kb long:

```
srec68 -p64 expl
```

tekhex68 Convert Program to Tektronix Tek Hex Format

SYNOPSIS

tekhex68 [*-options*] *prog*

DESCRIPTION

tekhex68 translates the program that is in the file named *prog*, and that was generated by the Aztec C68k/ROM linker into Tektronix Extended tekhex records. The program's extended tekhex records are written to one or more files, each of which contains the hex code for one ROM chip.

The ROM chips that are generated from the **tekhex68** output files will contain the program's code, followed by a copy of its initialized data.

Note: When a ROM system is started, its RAM contains random values; the Aztec C68k/ROM startup routine sets up its initialized data area, using the copy that is in ROM.

tekhex68 assumes that the size of each ROM chip is 2kb. You can explicitly define the size of each ROM using **tekhex68**'s **-p** option.

tekhex68 can also generate a symbol table file, in extended tekhex format, for a program. This is discussed below.

The Output Files

Even- and Odd-addressed Bytes in the Same Chips

tekhex68 can optionally generate extended tekhex records so that the program's even-addressed bytes are in one set of ROM chips, and its odd-addressed bytes are in another. This option is discussed below. In this section we discuss the output files that are created when this option is not used; i.e. when a program's even- and odd-addressed bytes are in the same set of ROM chips.

When neither **-e** nor **-o** is specified, **tekhex68** derives the name of each output file from that of the input file, by appending an extension of the form **.t***nn*, where *nn* is a number. For example, if the name of the linker-generated file is `prog`, then the names of the output files generated by **tekhex68** are `prog.t00`, `prog.t01` and so on, where the `.t00` file contains the extended tekhex records for the lowest-addressed ROM, `.t01` the extended tekhex records for the next ROM, etc.

For example, suppose that **tekhex68** is creating extended tekhex records for a program whose code and copy of initialized data will reside in three 2kb ROMs that begin at location **0**. Then **tekhex68** will create the following files:

prog.t00 Contains the extended tekhex records for the ROM chip that occupies addresses **0-0x7ff**;

prog.t01 Contains the extended tekhex records for the ROM that occupies **0x800-0xfff**;

prog.t02 Contains the extended tekhex records for the ROM that occupies **0x1000-0x17ff**.

Even- and Odd-addressed Bytes in Separate Chips

To place a program's even-addressed bytes in one set of ROM chips and its odd-addressed bytes in another, you must run **tekhex68** twice: once using the **-e** option to generate the extended tekhex records for the chips that contain the even-addressed bytes, and once using the **-o** option to generate extended tekhex records for the chips that contain the odd-addressed bytes.

When either **-e** or **-o** is specified, **tekhex68** generates one or more files, each of which contains the extended tekhex records for one ROM chip. By default, the size of each chip is 2kb, but you can use the **-p** option to explicitly define the chip size.

When the **-e** option is specified, the extension of the files are of the form **.e*nn***, where *nn* is a decimal number. The **.e00** file contains the extended tekhex records for the first of the ROM chips that contain even-addressed bytes, the **.e01** file contains the extended tekhex records for the second ROM chip, and so on.

When the **-o** option is specified, the extension of the files are of the form **.o*nn***, where *nn* is a decimal number. The **.o00** file contains the extended tekhex records for the first of the ROM chips that contain odd-addressed bytes, the **.o01** file contains the extended tekhex records for the second ROM chip, and so on.

Generating an Extended tekhex Symbol Table

To have **tekhex68** generate an extended tekhex symbol table file for a program, link the program with the **-g** option. This causes the linker to generate a file of debugging information for the program; this file's name is the same as that of the program file, with extension **.dbg**. If **tekhex68** finds a **.dbg** file for a program, it will automatically generate an extended tekhex file containing symbolic information. The extension of this extended tekhex file is **.tnm**.

The Options

tekhex68 supports the following options:

-b*x* The program begins *x* bytes into the first ROM chip, where *x* is a hexadecimal number. If this option is not specified, the program begins at the beginning of the first ROM chip.

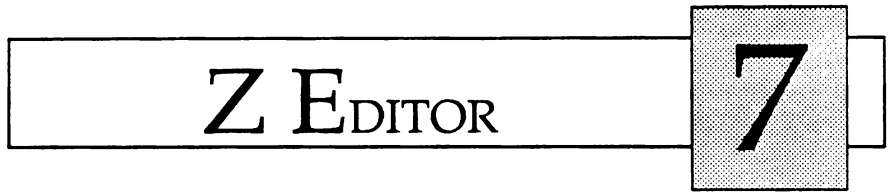
-e Output extended tekhex records for the program's even-addressed bytes.

-o Output extended tekhex records for the program's odd-addressed bytes.

-p*n* The size of each ROM is *n* kb, where *n* is a decimal number. If this option is not specified, the size defaults to 2kb. For example, the following command specifies that each ROM chip is 64kb long:

```
tekhex68 -p64 exmpl
```

-d Output a carriage return-line feed sequence after each extended tekhex record.



Chapter 7 - Z Editor

Z is a text editor for creating source programs, usually in the C programming language. Z has the following features:

- Similarity to the UNIX editor Vi: If you know Vi, you know Z.
- Full-screen editor: The screen acts as a window into the file being edited.
- A wealth of commands, specified with only a few keystrokes, that allow you to edit quickly and efficiently. The simple and natural way of entering commands and the mnemonic assignment of commands to keys make the commands easy to remember and use.
- Commands for the following:
  - ❑ Bringing different sections of a file into view
  - ❑ Inserting text
  - ❑ Making changes to text
  - ❑ Rearranging text by moving blocks of text around and by inserting text from other files
  - ❑ Accessing files
  - ❑ Searching for character strings and "regular expressions"

- Several commands that are useful for editing C programs, including commands for finding matching parentheses, square brackets, and curly braces; for finding the beginning of the next or preceding function; and for finding the next or preceding blank line.
- Most commands can be easily executed repeatedly.
- Sequences of commands, called macros, can be defined and executed one or more times.
- Changes are made to an in-memory copy of a file; the file itself is not changed until a command is explicitly given.
- Editing feature that allows the operator to request that a file containing a certain function be edited—**Z** finds the file and prepares it for editing.

Requirements

The maximum file size that you may edit using **Z** must not exceed your system's total memory size, *minus 64K*. In other words, to use Z editor, you must have at least 64K of available memory above and beyond the file you want to edit.

As you begin editing, **Z** will initially allocate a 16K edit buffer; **Z** will allocate additional 16K buffers when necessary.

Components

The **Z** package contains two programs:

- **Z** - the text editor
- **ctags**, - a utility for creating a file that relates tags to copies of C source files

Getting Started

Z is a very powerful tool for creating and editing C source programs, but its wealth of commands and options can be overwhelming to someone not familiar with it. This chapter gets you using Z as quickly as possible by presenting a small subset of the Z commands with which you can create and edit programs. Then, with the ability to create and edit programs, you can continue reading the rest of this chapter at your leisure to learn about the other features and commands of Z.

The first part of this chapter describes how to create a new C program, and the second part, how to edit an existing program.

Creating a New Program

You start Z by entering:

```
z hello.c
```

where **hello.c** represents the file to be edited. Since we are creating a new program, the file does not yet exist, therefore, Z displays a message on its status line (which may be either the first or last line of the display, depending on the system on which Z is running). On systems that use the first display line for status information, the screen looks like this:

```
"hello.c" No such file or directory
...
```

with the cursor on the left-hand column of the second line. On systems that use the last display line for status information, the screen looks like this:

```
...
"hello.c" No such file or directory
```

with the cursor on the left-hand column of the first line.

Z is now waiting for you to enter a command.

The Screen

As mentioned above, Z uses the one line of the display for displaying information and for echoing the characters of some commands that are entered.

The rest of the lines on the screen display the text of the file being edited.

The tilde characters on the screen lines tell you that Z has reached the end-of-file. These characters are not actually in the file.

Modes of Z

Z has two modes: command and insert, that allow you to enter commands and to insert text, respectively.

Insert Mode

With **Z** in insert mode, characters that you type are entered into a memory-resident buffer; the characters do not appear in the file until you exit insert mode and explicitly issue a command that causes **Z** to write the buffer to the file.

Z has several commands for entering insert mode; the one we want to use, **i**, allows text to be entered before the cursor. Type **i**. Notice that **Z** does not echo this command on the screen; it only does that for a few commands. Notice also that you are in insert mode, as evidenced by the message

```
<INSERT>
```

on the right-hand side of the status line.

You may now enter a program, just as you would on a typewriter. Notice that the cursor is positioned where the next character will be entered. Enter the **hello world** program:

```
main()
{
   printf("hello, world\n");
}
```

When you press the <CR> key after entering the **printf** line, the cursor was left positioned on the next line of the screen underneath the first nonwhite space character of the preceding line. This feature, "autoindent," is useful when creating C programs, encouraging statements within a compound statement to be indented and lined up. Autoindent can be disabled and enabled, and we will show you how later.

We want the closing curly brace of the main function to be on the first column of the line, not indented. So after you type the semicolon and then return at the end of the **printf** line, type the backspace key to get back to the first column, and then type the "}" key.

The backspace key can also be used to backspace over characters that you incorrectly type.

During insert mode, if you hold down the control key and type a **W**, the previous word typed is deleted.

When you have finished inserting the program, hit the escape key toexit insert mode and return to command mode. The key used as the escape key varies from system to system.

Exiting Z

To write the program you have just entered from the text buffer to the disk file **hello.c** and then exit **Z**, type **ZZ**. (Note that the "ZZ" must be typed as *uppercase*; the entry "zz" will not work.)

Occasionally you may want to exit **Z** without writing the text you have entered to a file; in this case, you would type

```
:q!
```

followed by a carriage return <CR>.

Editing an Existing File

The following describes the commands you need to make changes to an existing file.

Starting and Stopping Z

You get in and out of **Z** when editing an existing file just as you do when creating a new file. To start **Z**, enter

```
z hello.c
```

where **hello.c** is the name of the file to be edited.

Z reads the specified file into the text buffer, displays the first screen of file text, displays the file's statistics (name, number of lines, number of characters) on the status line, positions the cursor at the first character of the first line, and enters command mode, waiting for you to enter a command.

To stop **Z** and save the changes you have made, put **Z** in command mode and enter:

```
ZZ
```

(Again, note that the "ZZ" must be typed as *uppercase*; the entry "zz" will not work.) **Z** knows if you made changes to the original text or not; if you did, **Z** saves the original file by changing the extension of its name to .bak and then writes the modified text to a new file having the specified name. If a *.bak* file with that name already exists it will be deleted before the rename occurs.

If you did not make any changes, the **ZZ** command causes **Z** to halt without changing any disk files.

The command **:q!** quits **Z** without writing anything to the file being edited.

The Cursor

Before describing the commands for viewing and changing the text in Z's memory-resident buffer, we need to discuss the cursor.

In Z, the character position in the text that is pointed to by the cursor acts as a reference point: Most commands perform an action relative to that position. For example, the i command, described in the last section, allows you to enter text before the cursor. And the x command, to be discussed, deletes the character at which the cursor is located.

Therefore, we describe two types of commands in this chapter: those that move the cursor around in the text, thus bringing different sections of text into view, and those that modify text in the vicinity of the cursor.

Moving Around in the Text: Scrolling

The text you created for the `hello, world` program easily fit on a single screen. But most text files are too large to be viewed all at once, so we need commands to bring different sections into view.

Two such commands are the "scroll" commands: "scroll down," represented by the character Control-D, and "scroll up," represented by Control-U. That is, to execute the "scroll down" command, you hold down the control key and then press the D key. The rest of this manual refers to control characters using notation of the form ^D rather than Control-D, for brevity. Thus, the scroll up and scroll down commands are represented as ^U and ^D, respectively.

A scroll command moves the screen up or down in the file, bringing another half-screen of text into view. It is as if the text were on a reel of tape and the screen is a viewer: Scrolling down moves the viewer down the reel, and scrolling up moves the viewer up the reel.

When scrolling, the cursor will be left on the same position within the text after the scroll as before, if that position is still within view. Otherwise, the cursor is moved to a line in the text which was newly brought into view.

Moving Around in the Text: the Go Command

Scrolling is one way to move around in the text, but it is slow. If we have a large text file to which we want to append text, it would take a long time and many scroll commands to reach the end.

The **go** command, **g**, is one way to move rapidly to the point of interest in the text: Entering **g** by itself will move the cursor to the end of the text and, if necessary, redraw the screen with the text which precedes it.

The **g** command can also be preceded by the number of the line of interest; the cursor will move to the beginning of that line. So to move back to the first line of text, enter:

```
1g
```

The **g** command can be used to move to any line within the text, but since you usually do not know the numbers of the lines, the **g** command is mainly used to move to the beginning and end of the text.

Moving Around in the Text: String Searching

So, scrolling allows us to take a casual stroll through text, and the **g** command to move rapidly to the beginning and end of the file. What we need is a command to rapidly move to a specific point in the middle of the text.

The string search command, /, is such a command. When you enter /, followed by the string of interest, followed by a carriage return, **Z** searches forward in the text from the cursor position, looking for the string. If **Z** reaches the end of the text without finding the string, it will wrap around and continue searching from the beginning of the text.

If the string is found, the cursor is positioned at its first character and, if necessary, the screen is redrawn with its surrounding text.

If the string is not found, a message saying so is displayed on the status line of the screen and the cursor is not moved.

While the string search command and its string are being entered, the characters are displayed on the status line, and normal editing operations can be used, such as backspacing over mistyped characters.

Z remembers the last string searched for. To repeat the search, enter the find next string command, **n**.

Finely Tuned Moves

With the commands presented up to now, you can move to the area of interest in the text. The next few paragraphs present commands that move the cursor from somewhere within the area of interest to a specific character position from which changes will be made.

Some commands for this, from the many available in **Z**, are:

-	Moves the cursor **up** one line, to the first nonwhitespace character on the line.
CR (carriage return)	Moves the cursor **down** one line, to the first nonwhitespace character on the line.

space Moves the cursor **right** on the line on which the cursor is located.

backspace Moves the cursor **left** on the line on which the cursor is located.

These commands can be preceded by a number, which cause the command to be performed the specified number of times. For example,

3- Moves the cursor up three lines.

5<space> Moves the cursor right five characters. Note that *<space>* represents the space bar.

Deleting Text

You now have a repertoire of commands that allows you to move the cursor fairly quickly to any location in a text file. We are ready to move on to a few commands for modifying the text.

Two such commands for deleting text are DELETE CHARACTER, **x**, and DELETE LINE, **dd**:

x Deletes the character under the cursor.

dd Deletes the entire line on which the cursor is located.

Each of these commands can be preceded by a number, causing the command to be repeated the specified number of times. For example,

2x Deletes two characters.

3dd Deletes three lines.

More Insert Commands

You already know one command for inserting text: **i**, which allows text to be inserted before the cursor. We need a few more insert commands:

a Enters insert mode such that text is inserted following the cursor.

o (*Lowercase* o) Creates a blank line below the current line (i.e., the line on which the cursor is located), moves the cursor to the new line, and enters insert mode.

O (*Uppercase* o) Same as **o**, but the new line is above the current line.

Summary

With the set of commands presented in this chapter, you can edit any text file. You should continue reading this chapter to learn more about **Z**, while you use the basic command set for performing your editing chores.

You will find that **Z** has many more capabilities that allow you to perform functions more quickly and with fewer keystrokes than with the basic command set, and that allow you to perform functions that you cannot perform with the basic command set.

More Commands

This section describes the rest of the features and commands of **Z**, building and expanding on the information presented in the previous section.

The Screen

We have already discussed the basic details on **Z**'s use of the screen. Some of the more complex details concerning **Z**'s use of the screen are addressed below.

Displaying Unprintable Characters

A file edited by **Z** can contain any character whose ASCII value in decimal is less than 128, including unprintable characters, such as SOH (start of heading), LF (line feed), and ESC (escape). **Z** displays unprintable characters as two characters; the first is ^, and the second is the character whose ASCII value equals that of the character itself plus 0x40. For example, the unprintable character SOH is displayed as the pair of characters ^*A*, since the ASCII value of SOH is 1, and 1 plus 0x40 is 0x41, which is the ASCII value for the character "A".

Displaying Lines That Do Not Fit on the Screen

We have already stated that lines beyond the end of the file are displayed with the character ~ in the first column of the line on the screen. When you see the ~ character in the leftmost column of a line on the screen, this usually signifies that this line of the display does not contain a line of text. But lines that do not fit on the screen are displayed by **Z** in a similar manner.

Z allows lines to be entered that are longer than a screen line. Normally, **Z** simply displays such lines on several screen lines. In some cases, however, the entire line will not fit on the screen. For example, if the cursor is positioned at the beginning of the file, it may not be possible to display the text of an overly-large line at the bottom of the screen. In this case, **Z** displays an @ character in the first column of the screen lines on which the text would be displayed.

Thus, when you see the @ character in the leftmost column of a line on the screen, this signifies that the text that would have appeared on this line of the screen was too big, and not that the @ character is in the text.

Commands

When most commands are entered, **Z** does not echo the characters on the screen. However, two commands for which **Z** does display the characters on the screen are (1) those commands whose first character begins with ":" and (2) the string search commands.

For these commands, the characters are displayed on the screen status line, and can be backspaced over and reentered, if necessary. Also, **Z** does not act on such commands until you type the carriage return key, CR.

Special Keys

There are two keys that have special meaning for **Z**: the escape key, which is used to exit insert mode, and the control key, which is used in conjunction with another key to generate control characters. The actual keys used for these functions vary from system to system.

Paging and Scrolling

Previously, we described commands for scrolling through text, **^U** and **^D**. Another pair of commands allow you to page, instead of scroll, through text. The commands are **^B** and **^F**, which page backwards and forwards, respectively.

A page command brings the previous or next screen of text into view by redrawing the screen with the new text. Whereas scrolling was described as a viewer moving over a reel of tape, paging can be described as turning the pages of a book.

Paging moves you through text more quickly than scrolling does. However, since paging redraws the screen all at once, while scrolling changes it gradually, it is often more difficult to keep a sense of continuity when paging than when scrolling. As an aid to continuity when paging, two lines of text which were previously in view are still in view after paging.

Scroll commands can be preceded by a value specifying the number of lines to be scrolled up or down. If a number is not specified, the last scroll value entered is used; if a scroll value was never entered, it defaults to half a screen's worth of lines. Separate values are maintained for scrolling up and for scrolling down.

The scrolling and paging commands necessarily move the cursor within the text, but they cannot be used to home the cursor to an exact position at which changes are to be made. For this, you will have to use commands described in subsequent sections.

Searching for Strings

As stated, **Z** uses the / string search command to scan forward looking for a string. This section discusses additional searching capabilities that **Z** provides, the capability of **Z** to match patterns called REGULAR EXPRESSIONS, and special characters that are used to match a class of characters.

Additional String-Searching Commands

The other string-searching commands are:

?	Similar to /, but **Z** searches backwards, rather than forward, to find the previous occurrence of the string.
n	Repeats the last string-search command.
N	Repeats the last string-search command, but in the opposite direction.
:se ws=0	Turns the wrap scan option off.
:se ws=1	Turns the wrap scan option on.

When **Z** reaches the end or beginning of text without finding the string of interest, it normally wraps around to the opposite end of the text and continues the search. It does this because by default the wrap scan option is on. This option can be disabled by entering the set option command:

```
:se ws=0
```

thus causing the search to end when it reaches the end of text. The option can be reenabled by entering:

```
:se ws=1
```

Note that for this colon command, as for all colon commands, carriage return must be typed before the command is executed.

Regular Expressions

The strings you tell **Z** to search for are actually regular expressions, similar to the expressions or "patterns" that are used by the **grep** utility when matching strings. A regular expression is a pattern that is matched to character strings. The pattern can define a specific sequence of characters that make up the string; in this case, only that specific string matches the pattern. The pattern can also contain special characters that match a class of characters; in this case, the pattern can match any of a number of character strings.

For example, one such special construct is square brackets surrounding a character string; this matches any character in the enclosed string. So the regular expression

```
ab[xyz]cd
```

matches the strings

```
abxcd
abycd
abzcd
```

Another special character is *, which matches any number of occurrences of the preceding pattern. For example, the regular expression

```
ab*c
```

matches many strings, including

```
abc
abbc
abxyzc
```

and so on. And the pattern

```
ab[xyz]*cd
```

matches many strings, including:

```
abcd
abxcd
abxycd
abzzxcd
```

and so on.

The complete list of special characters and constructs that can be included in regular expressions is:

^	Matches the beginning of the line when it is the first character of a pattern.
$	Matches the end of the line when it is the last character of a pattern.
.	Matches any single character.
<	Matches the beginning of a word.
>	Matches the end of a word.
[*str*]	Matches any single character in the enclosed string.
[^*str*]	Matches any single character *not* in the enclosed string.

[*x-y*] Matches any character between x and y.

* Matches any number of occurrences of the preceding pattern.

Enabling Extended Pattern Matching

With **Z**, you can toggle the extent of pattern matching which will be done by **Z**. To have full pattern matching, type

:se ma=1

If you do not want full pattern matching, type

:se ma=0

which will only allow you to use ^ and $ in regular expressions.

By default, extended pattern matching is disabled.

Local Moves

This section presents more commands for moving the cursor fairly short distances: up or down a few lines, along the line on which it is located, and so on. Some commands to accomplish this that we have already discussed are the CR (carriage return), space, and backspace. The commands introduced here reflect the importance of finely tuned, quickly executed movements.

Moving Around on the Screen:

Here are some commands for moving the cursor short distances:

h Moves to the left one character.

j Moves down one line, leaving the cursor in the same column.

k Moves up one line, leaving the cursor in the same column.

l Moves right one character.

The keys **^H**, **^J**, **^K**, and **^L** are synonyms for **h**, **j**, **k**, and **l**, respectively.

These commands can be preceded by a number that specifies the number of times the command is to be repeated.

Z has commands for moving the cursor to the top, middle, and bottom of the screen; they are **H**, **M**, and **L**, respectively. The cursor is positioned at the beginning of the line to which it is moved.

Remember the - command, which moved the cursor up a line, to the first nonwhitespace character? As you might expect, + moves the cursor down a line, to the first nonwhitespace character. + is thus equivalent to CR.

Moving within a Line

The following commands have been discussed previously:

h, ^H, backspace	Left one character.
l, ^L, space	Right one character.

The following are a few more commands that allow you to move around on the current line:

^	Moves the cursor to the first nonwhitespace character on the line.
0	Moves the cursor to the first character on the line.
$	Moves the cursor to the last character on the line.

A few commands fetch another character from the keyboard, search for that character, beginning at the current cursor location, and leave the cursor near the character:

f	Scan forward, looking for the character, and leave the cursor on it.
t	Same as **f**, but leave the cursor on the character preceding the found character.
F	Same as **f**, but scan backwards.
T	Same as **t**, but scan backwards.
;	Repeat the last **f**, **t**, **F**, or **T** command.
,	Repeat the last **f**, **t**, **F**, or **T** command in the opposite direction.

Finally, the command | moves the cursor to the column whose number precedes the command. For example, the following command moves the cursor to column **56** on the current line:

```
56|
```

Word Movements

Z has several commands for moving the cursor to the beginning or end of a word that is near the cursor:

w	Moves to the beginning of the next word (alphanumeric only).

b Moves to the beginning of the previous word (alphanumeric only).

e Moves to the end of the current word (alphanumeric only).

For the preceding commands, a word is defined in the normal way: a string of alphabetical and numerical characters surrounded by whitespace or punctuation. There is a variant of each of these commands, differing only in the definition of a word: They think that a word is any string of nonwhitespace characters surrounded by whitespace. The variant of each of these commands is identified by the same letter, but in uppercase instead of lowercase:

W Moves to the beginning of the next word (any characters surrounded by whitespace).

B Moves to the beginning of the previous word (any characters surrounded by whitespace).

E To the end of the current word (any characters surrounded by whitespace).

Each of these commands can be preceded by a number, specifying the number of times the command is to be repeated. For example,

5w Moves forward five words.

The word movement commands cross line boundaries, if necessary, to find the word they are looking for.

Moves within C Programs

Z has several commands for moving the cursor within C programs:

]] and [[Moves to the opening curly brace, {, of the next or previous function, respectively.

% Moves to the parenthesis, square bracket, or curly bracket that matches the one on which the cursor is currently located.

{ and } Moves to the preceding or next blank line.

The [[and]] commands assume that the opening and closing curly braces for a function are in the first column of a line, and that all other curly braces are indented.

As an example of the % command, given the statement:

```
while (((a = getchar()) != EOF) && (c != 'a'))
```

with the cursor on the parenthesis immediately following the while, the % command will move the cursor to the last closing parenthesis on the line.

Marking and Returning

Z has commands that allow you to set markers in the text and later return to a marker. Twenty-six markers are available, identified by the alphabetical letters.

Unlike the other commands described in this section, these commands are not limited to moves within the current area of the cursor—they can move the cursor anywhere within the text.

A marker is set at the current cursor location using the command

m*x*

where *x* is the letter with which you want to mark the location.

There are two commands for returning to a marked position:

`` `x `` Moves the cursor to the location marked with the letter *x*

`'x` Moves the cursor to the first nonwhitespace character on the line containing the *x* marker.

Occasionally, you may accidently move the cursor far from the desired position. There are two single quote commands for returning you to the area from which you moved:

`''` Returns the cursor to its exact starting point.

`'` Returns the cursor to the first nonwhitespace character on the line from which the cursor was moved.

For example, if the cursor is on the line:

```
if (a >= 'm' && a <= 'z')
```

at the character <, then following a command which moves the cursor far away, the command `` `` `` will return the cursor to the < character, and the command `''` will return it to the beginning of the word `if`.

Adjusting the Screen

The Z command is used to redraw the screen, with a certain line at the top, middle, or bottom of the screen.

To use it, place the cursor on the desired line, then enter the Z command, followed by one of these characters:

CR To place the line at the top of the screen.

`-` To place it at the bottom.

`.` To place it in the middle of the screen.

The Z command is not a true cursor motion command, because the cursor is in the same position in the text after the command as before.

Control- L repaints the screen.

Making Changes

The previous section described the cursor movement commands. The next several sections describe commands for making changes to the text.

Small Changes

This section describes several more small commands. The first two commands listed below were already discussed in a previous section.

x	Which deletes the character at which the cursor is located.
dd	Which deletes the line at which the cursor is located.
X	Delete the character which precedes the cursor. Can be preceded by a count of the number of characters to be deleted.
D	Delete the rest of the line, starting at the cursor position.
rx	Replace the character at the cursor with **x** .
R	Start overlaying characters, beginning at the cursor. Type the escape key to terminate the command.
s	Delete the character at the cursor and enter insert mode. When preceded by a number, that number of characters is deleted before entering insert mode.
S	Delete the line at the cursor and enter insert mode; when preceded by a count, that number of lines is deleted before entering insert mode.
C	Delete the rest of the line, beginning at the cursor, and enter insert mode.
J	Join the line on which the cursor is positioned with the following line.

Operators for Deleting and Changing Text

Z has a small number of commands for modifying text. They all have the same form, consisting of a single letter command, optionally preceded by a count and always followed by a cursor mo-

tion command. The count specifies the number of times the command is to be executed. The command affects the text from the current cursor position to the destination of the cursor motion command, if the starting and ending position of the cursor are on the same line. If these positions are on different lines, the command affects all lines between and including the lines which contain the starting position and ending positions.

In this section, we are going to describe the operators for deleting and changing text, **d** and **c**:

d Deletes text as defined by the cursor motion command.

c Same as **d**, but **Z** enters insert mode following the deletion.

For example,

dw Deletes text from the current cursor location to the beginning of the next word.

3dw Deletes text from the cursor to the beginning of the third word.

d3w Same as **3dw** .

db Deletes text from the current to the beginning of the previous word.

d`a Deletes text from the cursor to the marker **a** , if the marker and the starting cursor position are on the same line. Otherwise, deletes lines from that on which the cursor is located through that on which the marker is located.

d/var Deletes text either from the cursor to the string **var** or between the lines at which the cursor is currently located and that on which the string is located.

d$ Deletes the rest of the characters on the line, and hence is equivalent to **D**.

Deleting and Changing Lines

Previously, we presented a command for deleting lines: **dd**. As you can see now, this is a special form of the **d** command, because the character following the first **d** is not a cursor motion command.

For all the operator commands, typing the command character twice will affect whole lines. Thus, typing cc will clear the line on which the cursor is located and enter insert mode. Preceding cc with a number will compress the specified number of lines to a single blank line and enter insert mode on that line.

Moving Blocks of Text

When text is deleted using the **d** or **c** command, it is moved to a buffer called the unnamed buffer. (There are other buffers available, which have names. More about them later).

Data in the unnamed buffer can be copied into the main text buffer using one of the **put** commands:

p Copies the unnamed buffer into the main text buffer, after the cursor.

P Same as **p**, but the text is placed before the cursor.

Thus, the delete and put commands together provide a convenient way to move blocks of text within a file.

The contents of the unnamed buffer is very volatile: When any command is issued that modifies the text, the text which was modified is placed in the unnamed buffer. This is done so that the modification can be undone, if necessary, using one of the undo commands. For example, if you delete a character using the **x** command, the deleted character is placed in the unnamed buffer, replacing whatever was in there. The unnamed buffer only holds the contents of the last command executed. So you have to be careful when moving text via the unnamed buffer. If you delete text into the unnamed buffer, expecting to place it elsewhere, then issue another command which modifies the unnamed buffer before issuing the put command. The deleted text is no longer in the unnamed buffer.

As you will see, the named buffers can also be used to move blocks of text, and their contents are not as volatile.

Duplicating Blocks of Text: the Yank Operator

The yank operator, **y**, copies text into the unnamed buffer without first deleting it from the main text buffer. When used with the put command, it provides a convenient way for duplicating a block of text.

The **y** operator has the same form as the other operators: an optional count, followed by the **y** command, followed by a cursor motion command. The command yanks the text from the cursor position to the destination of the cursor motion command, if the starting and ending positions are on the same line. If they are on separate lines, a whole number of lines are yanked, from the cursor position through the point the cursor would be moved to by the cursor motion command. The text is yanked into the unnamed buffer.

For example,

yw Copies text from the cursor to the next word into the unnamed buffer.

y3w Copies text from the cursor to the beginning of the third word.

3yw Same as **y3w**.

y`a Copies text from the cursor location to the marker **a** into the unnamed buffer, if the two positions are on the same line. Otherwise, copies entire lines between and including those containing the two positions.

As a special case, the command **yy** will yank a specified number of whole lines. The command **Y** is a synonym for **yy**. For example,

yy Yanks the line at the current cursor location.

y3w Yanks three lines, beginning with the one at the cursor location.

Named Buffers

In addition to the unnamed buffers, **Z** has 26 named buffers, each identified by a letter of the alphabet, which can be used for rearranging text. Text can be deleted or yanked into a named buffer and put from it back into the main text buffer.

The advantage of these buffers over the unnamed buffer in rearranging text is that their contents are not volatile: When you put something in a named buffer, it stays there and will not be overwritten unexpectedly. Also, as you will see, the named buffers can be used to move text from one file to another.

To yank text into a named buffer, use the yank operator, preceded by a double quote and the buffer name, and followed by a cursor motion command. For example, the following will yank three words into the **a** buffer:

```
"ay3w
```

and the following yanks four lines into the **b** buffer, beginning with the line on which the cursor is located:

```
"b4yy
```

Text is deleted into a named buffer in the same way: The delete command is used, preceded by a double quote and the buffer name. For example, to delete characters from the cursor to the a marker into the **h** buffer:

```
"hd `a
```

The preceding command, when the source and destination cursor positions are on separate lines, will delete a number of whole lines into the **h** buffer, from that on which the cursor is initially located through that containing the destination position.

To delete ten lines into the **c** buffer:

```
"c10dd
```

Text in a named buffer is put back into the main text using the **put** commands **p** and **P**, preceded by a double quote and the buffer name. For example:

"ap puts text from the **a** buffer, after the cursor.

"zP puts text from the **z** buffer, before the cursor.

Moving Text between Files

The named buffers are conveniently used to move text from one file to another. First yank or delete text from one file into a named buffer; then switch and begin editing the target file, using the **:e** command:

:e *filename*

(More on this later). Then move the cursor to the desired position and put text from the named buffer.

Shifting Text

The shift operators, < and > , which are used to shift text left and right a tab stop, respectively.

For example,

```
>/str
```

shifts right one tab stop the lines from that on which the cursor is located through that containing the string **str**.

Following the standard operator syntax, repeating the shift operator twice affects a number of whole lines:

5<< Shifts five lines left.

>> Shifts one line right.

Undoing and Redoing Changes

Z remembers the last change you made, and has a command, **u**, which undoes it, restoring the text to its original state.

Z also remembers all the changes which were made to the last line which was modified. Another UNDO command, **U**, undoes all changes made to that line.

Finally, the period command, ".", reexecutes the last command that modified text.

Inserting Text

The following commands have been discussed:

a Append after cursor.

i	Insert before cursor.
o	Open new line below cursor.
O	Open new line above cursor.
C	Delete to end of line, then enter insert mode.
s	Delete characters, then enter insert mode.
S	Delete lines, then enter insert mode.

This section discusses the remaining commands for entering insert mode and describes some other features of insert mode.

Additional Insert Commands

The other commands for entering insert mode are:

A	Append characters at the end of the line on which the cursor is located. This is equivalent to **$a**.
I	Insert before the first nonwhitespace character on the current line. This is equivalent to **^i**.

Insert Mode Commands

Some editing can be done on text entered during insert mode, using the following control characters:

backspace	Delete the last character entered.
^H	Same as "backspace" character.
^D	Same as "backspace" character.
^X	Erase to beginning of insert on current line.
^V	Enter next character into text without attempting to interpret it.

^V is used to enter nonprinting characters into the text. For example, to enter the character Control-A into the text, type

```
^V^A
```

That is, hold down the control key, then type the **V** key, then the **A** key, then release the control key. As mentioned earlier, nonprinting characters are displayed as two characters: "^" followed by a character whose ASCII code equals that of the nonprinting character plus 0x40.

Autoindent

The **Z** autoindent option is useful when entering C programs. When you are in insert mode and type the carriage return key with the autoindent option enabled, the cursor automatically indents on the new line to the same column on which the first nonwhitespace character appeared on the previous line. This feature is useful for editing C programs because it encourages statements that are part of the same compound statement to be indented the same amount, thus making the program more readable.

Z autoindents a line by inserting tab and space characters at the beginning of a new line. If you do not want the lines indented that much, backspace over these automatically inserted tabs and spaces until you reach the desired degree of indentation.

The autoindent option can be selectively enabled and disabled using the set options command:

:se ai=0 Disables autoindent.

:se ai=1 Enables autoindent.

When **Z** is activated, autoindent is enabled.

Macros

Z allows you to define a sequence of commands, called a MACRO, and then execute the macro one or more times.

When a macro is defined to **Z**, it is placed in a special buffer, called the macro buffer, and then executed once. There are two ways to define a macro to **Z**: immediately and indirectly.

Immediate Macro Definition

An immediate macro definition is initiated by typing the characters

:>

Z responds by clearing the status line, displaying these characters on the line, and waiting for you to enter the sequence of commands.

As you enter the commands, **Z** displays them on the status line and enters them immediately into the macro buffer, hence the term IMMEDIATE MACRO DEFINITION.

If you make a mistake while entering commands, you can simply backspace and enter the correct characters.

To terminate the definition, type the carriage return key. **Z** then executes the sequence of commands in the macro buffer. The contents of this buffer are not altered by executing the macro, so you can reexecute the macro without reentering it, as described below.

Examples

The following macro advances the cursor one line and deletes the first word on the new line:

```
:>+dw
```

This macro contains two commands: **+**, which advances the cursor, and **dw**, which deletes the word beneath the cursor.

The next macro moves the cursor to the previous line and deletes the last character on the line:

```
:>-$x
```

It contains three commands: **-**, which moves the cursor to the previous line; **$**, which moves the cursor to the last character on that line; and **x**, which deletes the character beneath the cursor.

You can also insert text using a macro. You enter insert mode using one of the normal insert commands. The characters that follow the insert command on the macro line, up to a terminating escape character, are then inserted into the text. The escape character causes **Z** to return to command mode and continue executing commands in the macro that follow the insert command.

For example, the following macro advances the cursor to the next line, deletes the second word on the line, inserts the character string "and furthermore", and deletes the last word on the line:

```
:>+wdwiand furthermore <ESC>$bdw
```

The last macro contains the following commands:

+ Advances the cursor to the next line.

w Moves the cursor to the second word on the line.

iand furthermore <esc>
Inserts the text **and furthermore** . <ESC> stands for the escape key.

$ Moves the cursor to the last character on the line.

b Moves the cursor to the beginning of the last word on the line.

dw Deletes the word beneath the cursor.

Z also allows you to search for a string from within a macro. Enter the string search command in the macro (for example, /), followed by the string, followed by the ESC character. For example, the following macro moves the cursor to the word **Melinda** and deletes it:

```
:>/Melinda<ESC>dw
```

It contains the commands

/Melinda<ESC>	Moves the cursor to **Melinda** . <ESC> stands for the escape key.
dw	Deletes **Melinda**.

The following macro finds **Melinda** and replaces it with **John** :

```
:>/Melinda<ESC>cwJohn<ESC>
```

It contains the commands:

/Melinda <ESC>	Moves the cursor to **Melinda**
cwJohn <ESC>	Changes **Melinda** to **John**.

Indirect Macro Definition

The other way of defining a macro is to yank a line containing a sequence of commands from the main text buffer into a named buffer.

Commands for indirect macro definition are:

@x	Causes **Z** to move the contents of the ***x*** buffer to the macro buffer and then execute it once.
"xv	A synonym for ***@x*** .

Indirect macro definition of macros has several advantages over immediate definition: For one, if a macro defined immediately is incorrect, you have to reenter the entire macro. With an indirectly defined macro, you can edit the macro definition in the main text buffer and then move it back to the macro buffer.

Another advantage is that you can store several macros in the named buffers and easily reexecute a macro, without having to reenter it. With immediate definition, when a new macro is defined, the previously defined macro is lost and must be reentered to be reexecuted.

One difference between entering macros immediately and via the named buffer concerns the method for specifying the end of a search string and for exiting insert mode. With immediate definition, you do this by typing the ESC key directly. For indirect definition, in which the macro is first entered into the main text buffer, typing the ESC key would cause **Z** to exit insert mode, not to enter the ESC key into the text of the macro. In this case, you enter the ESC key by first

typing Control-V, then ESC. This causes **Z** to enter the ESC character into the text of the macro and remain in insert mode.

Reexecuting Macros

Once a macro is defined and is in the macro buffer, it can be reexecuted by typing one of the commands:

@
v

Preceding the command with a count causes the macro to be executed the specified number of times.

Wrapping Around During Macro Execution

While executing a macro, **Z** may reach the beginning or end of the text, and want to continue beyond that point. This is especially true when reexecuting macros. The macro wrap option, **wm**, specifies whether **Z** should terminate the macro execution at that point, or continue at the opposite end of the text.

This option is enabled and disabled using the set options command:

:se wm=0 Disables macro wrapping.

:se wm=1 Enables macro wrapping.

When **Z** starts, this option is enabled.

Ex-like Commands

The **SUBSTITUTE** and **REPEAT LAST SUBSTITUTE** commands are a set of commands in the **Z** editor that are similar to commands in the UNIX Ex editor. This section describes the syntax for these commands, and then gives details about the substitute and repeat last substitute commands.

The ex-like commands consist of a leading colon, followed by zero, one, or two addresses identifying the lines to be affected by the command, followed by a single-letter command, followed by command parameters, and terminated by a carriage return. Most commands have a default set of lines that they affect, thus frequently allowing you to enter commands without explicitly specifying a range.

These commands support regular expressions, as defined in the **Z** documentation, for identifying addresses and strings to be searched for.

Addresses in Ex Commands

An address can be one of the following:

- A period, ., addresses the current line; that is, the line on which the cursor is located.
- The character $ addresses the last line in the edit buffer.
- A decimal number *n* addresses the *n*-th line in the edit buffer.
- *'x* addresses the line marked with the mark name *x*. Lines are marked with the **m** command.
- A regular expression surrounded by slashes (/) addresses the first line containing a string that matches the regular expression. The search begins with the line following the current line and continues toward the end of the edit buffer. If a line is not found when the end of the buffer is reached, and if the **Z** option **ws** is set to 1 (i.e., by the **:se ws=1** command), the search continues at the beginning of the buffer, stopping when the current line is reached.
- A regular expression surrounded by question marks (?) also addresses the first line containing a string that matches the regular expression. But in this case, the search begins with the line preceding the current line in the edit buffer and continues towards the beginning of the buffer. If a line is not found when the beginning of the buffer is reached, and if the **Z** option **ws** is set to 1 (i.e., by the **:se ws=1** command), the search continues at the end of the buffer, stopping when the current line is reached.
- An address followed by a plus or a minus sign, which in turn is followed by a decimal number *n* addresses the *n*-th line following or preceding the line identified by the address.

When two addresses are entered to define the range of lines affected by a command, the addresses are usually separated by a comma. They can also be separated by a semicolon; in this latter case, the current line is set to the line defined by the first address, and then the line corresponding to the second address is located.

When no value is specified for the first address in an address range, it is assumed to be the current line or the first line in the buffer, depending on whether the second address was preceded with a comma or a semicolon. When no value is specified for the second address in an address range, it is assumed to be the last line in the buffer. Thus, if neither the beginning nor the ending address of a range is specified, the range consists of either all the lines in the buffer or the lines from the current through the last line in the buffer, depending on whether comma or semicolon is used to separate the unspecified addresses.

The Substitute Command

The substitute command has the following form:

:[*range*]s /*pat* /*rep* / [*options*]

where square brackets surround a parameter to indicate that the parameter is optional.

Z searches the lines specified by *range* for strings that match the regular expression *pat*, replacing them with the *rep* string. If *range* is not specified, only the current line is searched. When the command is completed, the cursor is left on the character following the last replaced string.

The c Option

Normally, **Z** automatically replaces a string that matches *pat*. Specifying **c** as an option causes **Z** instead to pause when it finds a matching string, ask if you want the string to be replaced, and make the replacement only if you give your permission.

The g Option

Z replaces only the first *pat*-matching string on a line. Specifying **g** as an option causes **Z** instead to replace all matching strings on a line. In this case, after **Z** replaces a string on a line, it continues searching for more strings on the line at the character following the replaced string.

An ampersand (&) in the replacement string *rep* is replaced by the string that matched *pat*. The special meaning of & can be suppressed by preceding it with a backslash, \.

A replacement string consisting of just the percent character (%) is replaced in the current substitution by the replacement string that was used in the last substitution. The special meaning of % can be suppressed by preceding it with a backslash, \.

Examples

```
:s/aBD/def/
```

Search the line on which the cursor is located for the string **aBD**; if found, replace it with the string **def**.

```
:1,$s/ab*c/xyz/
```

Search all lines in the edit buffer for strings that begin with **a**, end in **c**, and have zero or more **b**'s in between; replace such strings with **xyz**. On any given line, only the first occurrence of a string that matches the pattern is replaced.

```
:/{/;/}/s/for/while/c
```

Find the first line following the current line that contains a `{;` then find the first line following this line that contains a `}`. In the lines between and including these lines, search for the string **`for`**. For each such string, ask if it should be replaced; if yes, replace it with **`while`**.

The "&" (Repeat Last Substitute) Command

The & command has the form

:[*range*]&

where brackets indicate that the parameters are optional.

The & command causes the last substitute command to be executed again, using the same search pattern, replacement string, and options as were used in the previous command. The command searches the lines that are specified in the & command range; if *range* is not specified, the substitution is performed on only the current line.

For example, the following command says that the options are in the file **`zopt.cmd`**:

```
set ZOPT=zopt.cmd
```

Starting and Stopping Z

You already know how to start and stop Z. This section presents more information related to starting and stopping Z.

Starting Z

Previously, we said that Z was started by specifying the name of the file to be edited on the command line:

Z *filename*

You may also start Z without specifying a *filename* or by specifying a list of files to be edited.

Starting Z without a Filename

Z can be started without specifying a filename, by entering the command:

```
Z
```

When you start Z without specifying *filename*, once it is active you normally tell Z the name of the file to be edited using the :e command:

```
:e filename
```

However, it is not necessary for Z to know the name of the file you are editing immediately upon opening: Z allows you to create and modify text in the text buffer without knowing the name of the file to which you intend to write the text. But you must explicitly tell Z to write the text to *filename* when you want to save the text that you worked on. You would use the command

```
:w filename
```

Z cannot automatically write the text, because it does not know which file you are editing.

The Option File

Z contains several options for controlling its operation in different situations, including the autoindent and macro wrap. (This manual contains a complete list of these commands at the end of this chapter.) This section presents another feature of Z related to options; i.e., the ability to set options automatically, when Z is started.

When Z starts, it reads options from the file specified by the ZOPT environment variable, if it exists.

The environment variable ZOPT defines the name of the options file.

Each line in the options file defines the value of one option, with a statement of the form

```
opt = val
```

where *opt* is the name of the option, and *val* is its value. For example, the following sets the tab-width option to eight characters:

```
ts=8
```

Setting Options for a File

When Z makes a file the edit file by reading it into the edit buffer, the file itself specifies the options to be in effect during its edit session. This feature is most useful in editing files that have different tab settings.

A file specifies option values by including strings of the form

:opt=val

in the first ten lines of the file. For example, the following line could be used near the front of a C program, causing a tab width of eight characters to be used:

```
/*  :ts=8   */
```

When Z starts editing a file, the tab width is set back to the default value, four characters, before the file is scanned for option settings.

Starting Z with a List of Files

You may start Z and pass it a list of names of files to be edited, as follows:

Z file1 file2 ...

Z remembers the list, and makes the first file in the list the edit file; that is, reads the file into the main text buffer and allows it to be edited.

Z contains a command, :n, that makes the next file in the list the edit file, after writing the contents of the text buffer back to the current edit file. File lists are discussed in more detail below.

Stopping Z

Previously, we presented the following commands for stopping Z:

ZZ If the file text in the edit buffer is modified, Z writes the text to the file, after changing the extension of the original file to **.bak**.

:q! Stops Z without writing the text to the file.

Two other commands for exiting Z are:

:wq Writes the text to the buffer, similar to, except that the text in the main text buffer is always written to the file, even if no changes have been made.

:q Conditionally stops Z. If no changes were made to the file text, Z stops; otherwise, it displays a message and remains active.

Accessing Files

This section discusses some additional commands for accessing files. For example, Z usually knows the name of the file you are editing, and in the sections that follow we will call this the edit

file. **Z** makes use of this knowledge, allowing you to write to the edit file without specifying it by name. For example, the **ZZ** command writes text to the edit file without requiring you to enter the name of the file.

Some commands allow you to access files without redefining **Z**'s idea of the edit file. The commands described in the next two subsections fall into this category.

Other commands cause **Z** to terminate editing of one file and begin editing another; this new file becomes the edit file. The commands described in the other subsections of this section are of this type.

Filenames

In the **Z** commands that require a filename, you enter the name using the standard system conventions. However, some characters are special to **Z**:

#	Refers to the last edit file.
%	Refers to the current edit file.
\	Causes the next character to be used in the filename and not be interpreted.

To enter a filename that contains these characters, precede the special character with the character "\".

Writing Files

The command **:w** writes the contents of the main text buffer to a file, without redefining the identity of the current edit file. It has the following forms:

:w	Write to the current edit file.
:w ***filename***	Write to the specified file
:w ***filename***	Same as **:w** ***filename***, but the file is overwritten if it exists.

As with all colon commands, carriage return must be typed to cause **Z** to execute the command.

When entered without a filename, **:w** creates a new file having the name of the current edit file and writes the contents of the edit buffer to it. This form of the **:w** command is commonly used to periodically save text during a long edit session, as protection against possible system failures.

The option **bk** tells **Z** whether it should save the original edit file before creating a new one. If **bk** is 1, the original is saved, and if 0, it is not. **Z** saves the original file by changing its name to .bak. An existing **.bak** file is erased before the rename occurs.

When a filename is entered with the **:w** command, the text is written to that file if it does not already exist. If it does, nothing is written and **Z** displays a message on the status line; in this case you must use the **:w!** form of the command to overwrite the file.

The **:w!** command unconditionally writes the text to the specified file after truncating the file, if it exists, so that nothing is in it. Unlike the **:w** command that does not specify a filename, the **:w!** command does not save the original file as a **.bak** file.

Reading Files

The command

:r *filename*

merges one file with a file being edited, without redefining the identity of the edit file. It reads the contents of the specified file into the main text buffer, inserting the new text following the line on which the cursor is located. It does not alter text that is already in the edit buffer.

Editing Another File

The following commands cause **Z** to stop editing one file and begin editing another, which then becomes the edit file:

:e *filename*	Edit the specified file.
:e! *filename*	Edit the file, discarding changes to the current edit file.
:e	Reload the current edit file.
:e!	Reload the current edit file, discarding changes.
:e #	Edit the previous edit file.
^^	Synonym for **:e #.** (the command is control-^).

Z begins editing another file by erasing the contents of the main text buffer, resetting the tab width to four characters, redrawing the display with the first screenful of lines from the file, and setting the cursor at the first character in the text.

When switching to a new edit file, **Z** does not change the contents of the named and unnamed buffers. Thus, these buffers can be used to hold text that is to be moved from one file to another and to contain commonly used macros.

The command

:e *filename*

causes the specified file to conditionally become the edit file. The condition is that changes must not have been made to the text of the current edit file since it was last written to disk. If this condition is met, then the switch is made; otherwise, Z displays a message on the status line and nothing is changed: The identity of the edit file is the same, the contents of the edit buffer are not modified, and the options are not changed.

If Z does not let you switch edit files when you enter

:e ***filename***

and you want to save the changes to the current edit file, enter the sequence:

:w
:e ***filename***

You can unconditionally cause Z to begin editing a new file by entering:

:e! ***filename***

In this case, Z does not care whether or not you made changes to the current edit file since it was last written to disk; it begins editing the new file without changing the previous edit file.

Sometimes the text in the edit file may get hopelessly scrambled, and you want to get a fresh copy of the edit file contents. The command

:e!

specified without a filename does just that.

Z not only remembers the name of the current edit file you are editing; it remembers the name of the last file you edited as well. Z allows you to refer to this name using the character # in :e commands, thus providing a quick means to reedit the previous edit file:

:e #

causes the previous edit file to conditionally become the current edit file, and

:e! #

causes it to unconditionally become the edit file.

The command ^^ (that is, Control-^) is a synonym for **:e #**.

Z also remembers the position at which the cursor was located in the previous edit file, and when you begin reediting this file it sets the cursor back to this position.

File Lists

Z's file list feature is convenient to use when you have several files to edit. You pass Z a list of the files and begin editing the first one. When you are finished with one file, a command switches to the next file in the list, after you have explicitly saved the changes to the current edit file. An op-

tion to the command prevents Z from saving changes, and another command rewinds the file list so that you are back editing the first file in the list again.

There are two ways to pass the list of files to be edited to Z: as parameters to the command that starts Z, and as parameters to the :n command. In each case, Z remembers the list and makes the first file in the list the edit file. For example,

```
Z file1 file2 file3
```

starts Z and defines the list of files—**file1, file2,** and **file3**. Z makes **file1** the edit file; that is, prepares it for editing by reading it into the edit buffer and displaying its first lines.

When Z is active, the command

```
:n file4 file5 file6
```

defines a new list of files—**file4, file5** and **file6**. Z makes **file4** the edit file.

When used without a files list, the :n command switches from one file in the list to the next:

:n! Switches without writing anything to the current edit file.

The **:rew** command rewinds the file list, i.e., makes the first file in the list the edit file. This command behaves like the **:n** command, in that any change to the current edit file must be rewritten before rewinding; and when an exclamation mark is appended to the command, the rewind occurs, regardless of the state of the current edit file.

Tags

Z has a feature useful for editing large C programs that contain many functions distributed over several files. With the aid of a cross-reference file relating tags, (i.e., function names), to the files containing them, you simply tell Z the name of the function that you want to edit and Z makes the file containing it the edit file by reading it into the edit buffer and positioning the cursor to the function.

The following commands specify the tag of the function to be edited:

:ta *tag* — Position to the function named tag in the appropriate file, if the current edit file is up to date

:ta! *tag* — Same as **:ta** *tag*, but the switch to the new file occurs even if the current edit file is not up to date.

When using the **:ta** command, the current edit file is considered up to date if the text in the edit buffer has not been modified since it was last written to the file. When used without the trailing !, the **:ta** command does not switch edit files if the current edit file is not up to date; it only displays a message on the status line. You can then either write the text in the edit buffer to the file and reenter the **:ta** command, or immediately enter the **:ta!** command, to switch edit files anyway.

If *tag* ends up in the current file, it works regardless of the current file's modification status.

The command

```
^]
```

Control-], is convenient when, while editing or viewing one function, you want to edit or examine a function that it calls. You just set the cursor to the name of the called function and enter **^]** ; **Z** makes the file containing the called function the edit file, and positions the cursor to this function.

For example, while examining the file **crtdvr.c**, you may come across a call to the function **pcdvr**, and may want to take a look at it. By positioning the cursor at the beginning of the word **pcdvr** and typing **^]**, **Z** makes the file containing **pcdvr** the edit file and leaves the cursor positioned at this function.

The ctags Utility

The utility program **ctags** creates the cross reference file, **tags**, that relates function names to the file containing them. **ctags** is activated by a command of the form

ctags *file1 file2 ...*

where file1, ..., are names of files whose functions are to be placed in the cross reference file. A filename can specify a group of files using the character *. For example:

```
*.c
```

specifies all files whose extension is **.c**, and

```
f*.c
```

specifies all files whose first character is **f** and whose extension is **.c**.

ctags creates the cross reference file, **tags**, in the current directory on the default drive.

When a tags command is given, **Z** searches for this file in the current directory.

Executing System Commands

Z has two commands that allow you to execute system commands while **Z** is active and then return to **Z**:

:!*cmd*	Execute the system command *cmd*
:!!	Re-execute system command

For example,

```
:! dir *.c
```

executes the command **dir *.c** and then returns to **Z**.

Options

Z provides several options under user control that define how **Z** behaves in certain situations. Most of these options have been discussed peripherally in previous sections, when appropriate. This section focuses on the options.

Each option is identified by a code. The options and their codes are:

ai Auto-indent option. When this option is enabled and you begin inserting text on a new line, **Z** automatically indents the line by inserting tabs and spaces so that the text you type is correctly aligned with the text in the line above it. By default, this option is enabled.

eb Error bells option. When this option is enabled, **Z** beeps when you make a mistake. By default, this option is enabled.

ma Magic option. When this option is enabled, regular expressions used in string searches can include extended pattern matching characters. Otherwise, only the characters ^ and $ are special and the extended pattern matching constructs are gotten by preceding them with \. By default, this option is disabled.

ts Tab Set Option. Specifies the number of characters between tab settings. By default, the tab width is four characters.

wm Wrap On Macro Option. When this option is enabled, and a macro being executed reaches the end of the buffer, the macro wraps around to the beginning of the buffer and continues. By default, this option is enabled.

ws Wrap On Search Option. When this option is enabled and a search for a string reaches the end of the buffer without finding the string, the search continues at the opposite end of the buffer. By default, this option is enabled.

bk Defines whether **Z**, when a **:w** command is entered to write the edit buffer to the current edit file, should save the original edit file before creating a new one.

co Set the attribute byte of the screen's text area. For example, **:se co=7** sets the attribute byte of text area characters to 7 (white characters on black background). For a definition of screen attribute bytes, see the PC technical reference manual.

sc

Set the attribute byte of the screen's status line. For example, **:set sc=112** sets the status attribute byte of status line characters to 112 (black characters on white background).

43 Switch to 43-line mode (**:se 43=1**) or 25-line mode (**:se 43=2**).

sm Silent Macro Option. When this option is enabled, macros operate silently. If it is disabled, macros display their commands as they execute. The main advantage to silent operation is that it is faster.

An option is enabled by setting it to 1, and disabled by setting it to 0.

Differences Between Z and Vi

Z is very similar to the UNIX editor Vi, in the following ways:

- Both are full-screen editors, display text in the same way, and reserve one line of the display for messages;
- They have the same two modes: command and insert;
- Z supports most of the Vi commands. The Z commands are activated by the same keystrokes and perform the same functions as their Vi counterparts.

Z and Vi differ in the following ways:

- In Z, the buffer in which text is edited is entirely within RAM memory; in Vi, the buffer is both in memory and on disk. Because of this, Z is restricted in the size of program that can be edited, but Vi is not;
- A single copy of Vi can be configured to use any type terminal. A single copy of Z is preconfigured to use just one terminal;
- Vi has an underlying editor, ex, whose commands can be executed while Vi is active. Z does not have an underlying editor. However, Z does support some ex commands directly; these are the commands whose first character is ":". (Vi interprets the ":" as a request to execute the ex command which is entered after the ":");
- Vi has commands and options useful for editing documents and for editing LISP programs, but Z does not;
- With Vi, you can create a shell and suspend Vi while executing commands from within the new shell. With some Vis, you can also suspend Vi while executing commands from the shell that activated Vi. Z does not support either of these features;

- Vi saves the last nine deleted blocks of text and has commands with which it can recover them, if necessary. **Z** lets you recover the last deleted block;

- With Vi, operator commands can affect exactly the characters between the starting and ending cursor positions, even when the positions are on different lines. **Z** has variations of these commands which allow whole lines to be affected, between and including the lines containing the two positions.

  In **Z**, operator commands in which the starting and ending cursor positions are on different lines always affect whole lines, between and including the lines containing the two positions.

IBM PC Features

The following features are supported by the PC DOS version of the **Z** text editor.

- The PC's cursor motion keys move the cursor.

- The function keys cause macros to be executed.

These features are discussed in the following paragraphs.

Key Substitutions

When you type certain special keys on a PC keyboard, keys that normally do not have any meaning to **Z**, **Z** substitutes for these keys characters that do have a meaning to **Z**. The following table lists these special keys and the characters and the characters that are substituted for them. As shown in the table, for each special key up to three possible substitutions can be made:

- The **Normal** column indicates the substitutions when neither the shift nor the control key is being held down.

- The **Shift** and **Control** columns define the substitutions that are made when a special key is typed while the shift or control key is being held down, respectively.

Typed Key	Substituted Characters		
	Normal	Shift	Control
Home		lg	z/r
Up Arrow	k	H	
PgUp			[[
Left Arrow	h	B	b
5(keypad)		%	
End	$	G	z-
Down Arrow	j	L	
PgDn			]]
Ins	i	o	
Del	x	D	
- (keypad)	-	-	
+ (keypad)	+	+	

In this table, \r stands for "carriage return"; and ^ stands for "control key".

If you type one of the special keys while in insert mode, except when holding down the shift key, Z will return to edit mode and then execute the command that corresponds to the substituted characters. In the special case, that is, typing a special key with shift depressed while in edit mode, the appropriate character is entered into the edit buffer.

You can enable and disable the substitutions that are normally made when a special key is typed with the shift key depressed, by setting the option xt to 0 or 1, respectively (i.e., by entering **:se xt=1** or **:se xt=0**).

Function Key Macros

With the PC version of Z, macros containing up to 19 characters can be associated with function keys and then executed when the function key is typed. Up to four macros can be associated with a given function key: the macro that is executed depends on whether the shift, control, alternate, or none of these is depressed when the function key is typed.

You can execute a function key macro while in insert mode; in this case, Z will return to edit mode and then execute the macro.

To define the macro that is associated with a function key, enter a command of the form

:se ***xn=macro***

where *n* is the number of the function key. *x* is **s**, **c**, **a**, or **f**, depending on whether the macro is to be executed when the function key is typed in conjunction with the shift, control, or alternate key, or with none of them respectively.

For a list of the macros that are associated with the function keys, type **`:se all`**.

For your convenience, **Z**, when it starts, associates some commonly used commands with the function keys. This association is listed in the following table. You can of course redefine the function key macro association as described above. In this table, **`\r`** stands for the carriage return character.

Function Key	Associated Macros			
	Normal	Shift	Control	Alternate
F1	:x\r	:q!\r	:W\r	"a
F2	:!	:!!	:!*dir*	'a
F3	:>	@@		"b
F4	:se	:se ts=	:se ma=	'b
F5	:rew\r	:rew!\r		"c
F6	:ta	:ta!	^]	'c
F7	:e #\r	:e! #\r	:f\r	"d
F8	:e	:e!	:e\r	'd
F9	:n\r	:n!\r	:n	:se xt=0\r
F10	:w\r	:w!	:w	:se xt=1\r

Command Summary

Starting Z

z ***name*** edit file *name*

z name1 name2 ....

edit file *name1*, rest via **:n**

The Display

~lines	lines past end of file
@lines	lines that do not fit on screen
^x	control characters
tabs	expand to spaces, cursor on last

Options

ak	allows you to move the cursor via the keyboard arrow keys
ai={1|0}	auto-indent {on | off}
eb={1|0}	error bells {on | off}
ma={0|1}	magic {off | on}
ts=*val*	tab width (default is 4)
wm={1|0}	wrap on search when executing macro {on | off}
ws={1|0}	wrap on search scan {on | off}
bk={1|0}	save original file as **.bak** {on | off}
sm={1|0}	silent macro execution {on | off}

co=*val*	Set text attribute to *val*
sc=*val*	Set status line attribute to *val*
43={0|1}	Set 43-or 25-line mode

Adjusting the Screen

^F	forward screenful
^B	backward screenful
^D	scroll down half screen
^U	scroll up half screen
zCR	redraw, current line at top
z-	redraw, current line at bottom
z.	redraw, current line at center

Positioning within File

g	go to line (default is end of file)
G	go to line (default is beginning of file)
/pat	move cursor to *pat* searching forwards
?*pat*	move cursor to *pat* searching backwards
n	repeat last / or ?
N	repeat last / or ? in reverse direction
]]	next "^{"
[[	previous "^{"
%	find matching (), {}, or [].

Marking and Returning

``	previous context
''	first nonwhite at previous context
m*x*	mark position with letter *x*
'*x*	to mark *x*
'*x*	first nonwhite at mark *x*

Line Positioning

H	top of screen
M	middle of screen
L	bottom of screen
+	next line, first nonwhite
CR	next line, first nonwhite
-	previous line, first non-white
LF	next line, same column
j	next line, same column
^K	previous line, same column
k	previous line, same column

Character Positioning

0	beginning of line
^	first nonwhite at beginning of line
$	end of line
space	forward a character

^L	forward a character
l	forward a character
^H	backwards a character
h	backwards a character
f*x*	find character *x* forward
F*x*	find character *x* backwards
t*x*	position before character *x* forward
T*x*	position before character *x* backwards
;	repeat last f, F, t or T
,	repeat last f, F, t or T in reverse direction
|	move to specified column number

Words and Paragraphs

w	word forward
W	blank delimited word forward
b	back word
B	back blank delimited word
e	end of word
E	end of blank delimited word
}	to next blank line
{	to previous blank line

Insert and Replace

a	append after cursor

A	append at end of line
i	insert before cursor
I	insert before first non-blank in line
o	open line below current line
O	open line above current line
rx	replace single character with *x*
R	replace characters

Corrections During Insert

^H	erase last character
^D	erase last character
^X	erase to beginning of insert on current line
^V	insert following character directly
^W	delete previous word typed

Operators

d	delete
c	delete and insert
<<	left shift
>>	right shift
y	yank

Miscellaneous Operations

D	delete rest of line
C	change rest of line
s	substitute characters
S	substitute lines
J	join lines
x	delete characters starting at cursor
X	delete characters before cursor
Y	yank lines

Yank and Put

p	put after current
P	put before current
"*x*p	put from buffer *x*
"*x*y	yank to buffer *x*
"*x*d	delete to buffer *x*

Undo and Redo

u	undo last change
U	restore current line
.	repeat last change command

Macros

@X	execute macro in buffer *x*
"*x*v	execute macro in buffer *x*
@@	repeat last macro
v	repeat last macro

Colon Commands

:e *name*	edit file *name*
:e	re-edit last file
:e! *name*	edit file *name*, discarding changes
:e!	re-edit last file, discarding changes
:e #	edit alternate file
^^	edit alternate file
:e! #	edit alternate file, discarding changes
:fn	searches the file funclist or the environment variable FUNCLIST for a specified string; also invoked by typing ^_.
:r *name*	read file *name* into current file
:w	write back to file being edited
:wq	write back to file and quit
:w *name*	write to file *name* if does not exist
:w! *name*	write to file *name*, delete if exists
:q	quit
:q!	quit, discarding changes
:x	quit, saving file if modified

ZZ	quit, saving file if modified
:f	show current file and line
:f *name*	change *name* of current file
^G	show current file and line
:n	edit next file in list
:n!	edit next file in list, discarding change
:n *arg1 arg2....*	specify new list
:rew	point back to beginning of list
:rew!	point back to beginning, discarding changes
:ta *tag*	position to *tag* in appropriate file, searches file pointed to by environment variable TAGS if tags does not exist or *tag* is not found in it.
^]	same as **:ta** using word at cursor
:ta! *tag*	position to *tag*, discarding changes
:>macro	specify and execute immediate macro
:set *opt1=val opt2=val ...*	set editor options
:se *opt1=val opt2=val ...*	set editor options
:set all	display current option settings
:[*range***]s/***pat***/***rep***/[***options***]**	substitute *rep* format in *range*
:[*range***]&**	repeat last substitute command

Chapter 8 - Library Customization

This chapter discusses the customization of the library functions that are provided with Aztec C68k/ROM. It is organized into two sections: the first discusses changes that you might make to the libraries. The second discusses the actual generation of the libraries.

It is assumed that you have installed the source for the Aztec C68k/ROM functions in a set of sub-directories, as described in the **Tutorial** chapter. It is also assumed that you are using the **make** program that is provided with Aztec C68k/ROM.

For more information about the functions described in this chapter, see the **Library Overview** and **Library Functions** chapters.

Modifying the Functions

Many of the functions provided with this package will run, without modification, on any 68000-based system. Some, however, are system dependent and must be specially implemented for your system.

The functions that may need to be rewritten are:

- The startup function;
- The unbuffered I/O functions;
- The low-level heap allocation functions **brk()** and **sbrk()**;
- The exit functions **exit()** and **_exit()**.
- The time functions **time()** and **clock()**.

Modifying the Startup Routine

A program's startup routine is executed when the program is started. It performs program initialization and then calls the program's **main** function.

The source for the startup routine that is provided with Aztec C68k/ROM is in the file **rom68.a68**, in the **lib\rom68** directory. The supplied version of this routine makes the following assumptions about a program that contains it, and about a system that contains the program:

- The system's startup/reset vectors and interrupt vector table are in ROM.
- The program is the "startup program" of the system containing it. That is, the program will gain control on system startup or reset.
- The program's code and a copy of its initialized data are in ROM. It's the startup routine's duty to set up the program's initialized data area in RAM from the ROM copy.
- The startup routine is at the beginning of the program's code segment.
- The system doesn't support interrupts.

If these assumptions aren't satisfied by your system, you will have to modify the startup routine. The following paragraphs discuss changes that can be made for several types of programs and systems.

ROM-based, Interrupt-driven Systems

Since a system's memory must begin with startup vectors and be followed by the table of interrupt vectors, the above assumptions mean that the startup module must contain assembly language statements that pre-initialize these vectors. In fact, the supplied startup routine does contain statements that pre-initialize the startup vectors: the stack vector points to the top of the area that's reserved for the stack, and the code vector points to the **.begin** label in the startup module.

However, the supplied startup routine can't pre-initialize the interrupt vector table, since that's system dependent. The supplied startup routine simply reserves space for the table.

Thus, if a program that satisfies all the above assumptions is to be placed on a system that supports interrupts, you must modify the startup routine, replacing the statement that reserves space for the interrupt table with statements that pre-initialize the vectors for supported interrupts.

ROM-based, Non-startup Programs

If the startup routine will be included in programs that will be burned into ROM but that won't be a system's startup program, you can remove the statements in the startup routine that pre-initialize the system startup vectors and that reserve space for the interrupt table.

Most of the code in a program's startup routine needs to be executed just once. For example, its initialized data area in RAM needs to be set up from the copy in ROM just once; and its uninitialized data area needs to be cleared just once. So if a program will be called more than once, you could design your startup routine so that this special startup code is executed just once.

The advantages to this are:

- It speeds up interprogram calls
- Variables are preserved between interprogram calls.

To do this, you could have a second entry point into a program, in addition to the standard entry point. The first call is made to the standard entry point, and all subsequent calls are made to the secondary entry point.

The secondary entry point performs just those operations that need to be done on each entry to the program. For example, if the program uses the small code or small data memory model, the secondary entry point would save the contents of the small model support register and set it up for the called program.

To implement the two entry points, you could add two jump instructions to the beginning of the program's startup routine: the first jumps to the startup routine's **.begin** label; the second jumps to the secondary entry point code.

Systems Whose Interrupt Table Is In RAM

The interrupt vector table of some 68k systems must reside in RAM, to enable the program to dynamically set up and change the vectors. Since this table is normally in ROM, this requires special hardware and corresponding changes to the startup routine.

In this section first we describe why the interrupt table normally resides in ROM. We then present two hardware techniques used to place the table in RAM and the corresponding changes that must be made to the startup routine.

Why the Interrupt Table Is Normally In ROM

On a 68k system, the startup vectors occupy the first eight bytes of memory and the interrupt table follows. If the system uses a standard configuration (i.e. a typical microcomputer system configuration that doesn't use special hardware),then the startup vectors must be in ROM, so that they will be already initialized when the system is turned on or reset. Since the smallest ROM chip is about 2K bytes, this in turn means that the interrupt table of a standard 68k system must also be in ROM.

Solution 1: Move the Startup Vectors

One way to allow the interrupt table to reside in RAM is to move the startup vectors away from the interrupt table:

- Put RAM in the lowest-addressed section of memory, so that it extends at least from location 0 through the end of the interrupt table;
- Include the startup vectors in the code section of the system's startup program, at a fixed offset from the beginning of the program's ROM;
- Insert special hardware on the address bus between the processor and memory. On powerup or system reset, this hardware intercepts the processor's first two accesses of memory, which are requests by the processor for the startup vectors, and translates the accompanying addresses (i.e. locations 0 and 4) to those of the fields within ROM that actually contain the startup vectors.

To support this hardware configuration, you should remove the statement in the startup module that reserves space for the interrupt table and add executable code that initializes the table. To put the startup vectors at a fixed place in ROM memory, to which the special hardware can redirect attempts by the processor to access them, you could leave the statements that define the startup vectors in the startup routine and then link the startup routine as the program's first module. The startup vectors will then be in ROM, in the first eight bytes of the startup program's code section.

Solution 2: Move the Interrupt Table

Another way to put the interrupt table in RAM is to move the interrupt table away from the startup vectors:

- Put the RAM for the interrupt table in an unused section of the system's memory space, a section that is not near the low end of memory.
- Put the ROM that contains the code for the system's startup program in memory, beginning at location 0. The startup routine should be at the beginning of the program's code section; the only changes that it needs are executable statements that initialize the interrupt table.
- Put a programmable logic array on the address bus, between memory and the processor. This will intercept requests to access an interrupt vector (i.e. accesses of memory between locations 8 and 0x400) and translate the accompanying address to the address in RAM at which the vector is actually located.

Heap Set-up Code

The startup routine initializes variables that define the boundaries of a program's heap so that the heap occupies the area that you defined using the **+j** and **+s** linker options. By default, this area begins at the end of the program's unititialized data segment and is 2kb long. If this does not meet your needs, you can change these initializations.

These variables, which are used by the **sbrk()** and **brk()** functions, are:

_mbot	Points at the bottom of the heap.
_mtop	Points at the top of the heap.
_mcur	Points at the top of allocated heap space.

These are the names that a C-language module uses to access the variables; an assembly language module uses these names, with an additional prepended underscore (e.g. *__mbot*).

Rom-based Initialized Data

The startup routine contains statements that set up a program's initialized data segment in RAM from its copy in ROM. Remove these statement if the program's initialized data is to remain in ROM; i.e. if you linked the program without using the linker's **-d** option.

Ram-based Programs

If you are creating programs that won't be put into ROM (for example, programs that will run on a system that uses an operating system), here are some changes you may want to make to the startup routine:

- Remove the code that initializes the startup vectors and that reserves space for the interrupt table.

- Change the code that sets up the stack register. The operating system probably defines the area reserved for a program's stack (for example, on entry the stack register may already be initialized). If it doesn't, you could, for example, define space for the stack in the uninitialized data area, and point the stack register at the top of this area.

- Remove the code that moves the copy of initialized data from ROM to RAM.

- Change the code that initializes the pointers to heap space.

- The startup routine jumps directly to the **main** function. If you want your system to support the passing of arguments to the **main** function, you may want to have the startup routine call a C-language function, which gets the arguments (for example, getting them from the console) and then calls **main** .

A program created by Aztec C68k/ROM must always be loaded at an address that is defined when the program is linked. If you need to create programs whose load address is not known until the program is loaded, please contact Manx Software Systems Technical Support group.

Modifying the Unbuffered I/O Functions

Two classes of I/O functions are provided with Aztec C68k/ROM. The unbuffered I/O functions are system dependent, and the standard I/O functions call the unbuffered.

- The unbuffered I/O routines are system dependent, and must be specially written for your system;

- The standard I/O routines are system independent, but call the unbuffered I/O routines.

Thus, before your programs can access either the standard or unbuffered I/O routines, you must implement the unbuffered I/O routines.

The unbuffered I/O functions are:

close()	**creat()**	**ioctl()**	**isatty()**	**lseek()**
open()	**read()**	**remove()**	**rename()**	**unlink()**
write()				

Descriptions of the unbuffered I/O functions are in the **Library Functions** and **Library Overview** chapters. The following paragraphs present additional information that may be of use when writing your own versions of these functions.

File Descriptors

Associated with each file or device that is open for unbuffered I/O is a positive integer called a "file descriptor". A file descriptor is one of the parameters that is passed to an unbuffered I/O function; it defines the file or device on which the I/O is to be performed. There's usually a limited number of file descriptors, which of course limits the number of files and/or devices that can be simultaneously open for I/O.

When there's lots of files and devices...

If a system supports disk files and/or supports more devices than file descriptors, the file descriptors must be dynamically allocated. That is, before I/O with a file or device can begin, a function must be called that assigns a file descriptor to it; and when the I/O is done another function must be called to de-assign the file descriptor. In this case, a table is usually provided that has entries defining the status of each file descriptor and that is accessible to all the unbuffered I/O functions. Here's how the unbuffered I/O functions make use of the table:

- **open()** and **creat()** prepare a file or device for unbuffered I/O. They scan the table for an unused entry, and initialize the entry with information about the file or device. For example, the entry for an open device might contain the device's address; that for an open file might contain the file's current position and access mode. As the file descriptor for the opened file or device, **open()** and **creat()** return the entry's index into the table.

  open() and **create()** must allow a file or device to be opened in one of two modes: text or binary, as defined by the presence of O_TEXT or O_BINARY flag in the call to **open()** or **creat()**. For a file or device opened in text mode, **read()** and **write()** must perform end-of-line conversions. For example, **write()** might translate `\n` to `\r\n` and read might translate `\r\n` to `\n`.

  For a file or device opened in binary mode, these translations do not occur.

- **read()**, **write()**, **lseek()**, **ioctl()**, and **isatty()** perform operations on, and determine the status of, an open file or device. The file descriptor of the file or device is one of the parameters passed to them. They examine the file descriptor's table entry for information about the file or device.

- **close()** completes I/O to the open file or device having a specified file descriptor. Most of the operations that **close()** performs depend on the particular file or device; but it always marks the descriptor's table entry as being unused.

- **unlink()**, **remove()**, and **rename()** don't use the file descriptor table at all.

When only devices are supported...

If programs access just devices (i.e. not files), if there are fewer devices than file descriptors, and if your programs make limited use of the standard I/O functions (as defined below), you can simplify the unbuffered I/O functions by doing away with the file descriptor table, hard-coding

the assignment of devices and file descriptors into the unbuffered I/O functions, and leaving **open()**, **creat()**, and **close()** as mere stubs that simply return when called.

For example, you could code into the **write()** function the fact that file descriptor 5 is associated with a printer at a certain address. Then to write to the printer, a program could simply issue a call to **write()**, telling it to write to file descriptor 5. It wouldn't have to first call **open()** or subsequently call **close()**.

Pre-assigned file descriptors

By convention, file descriptors 0, 1, and 2 are pre-assigned to the system console, even when all other file descriptors are dynamically assigned. To perform an unbuffered I/O operation on the console, a program simply calls the appropriate function, specifying one of these file descriptors; it need not first call **open()** or subsequently call **close()**.

Some systems allow the operator to redirect file descriptors 0 and 1 to other files and/or devices, by specifying special operands on the command line that starts a program. This is done by inserting a special function between the startup routine and the user's **main()** function. If any redirection operands are found in the command line, this special function closes the specified file descriptor by calling **close()** and reopens it to the new file or device by calling **open()**. By convention, the command line operand to redirect file descriptor 0 consists of "" followed by the file or device name. The command line operand to redirect file descriptor 1 consists of "" or ">" followed by the file or device name. "" causes a new file to be created. ">" causes a file to be appended to, if it already exists, or to be created, if it doesn't exist.

Interaction of the standard I/O and unbuffered I/O functions

The standard I/O functions call the unbuffered I/O functions. Because of this, the standard I/O operations that a program will perform places implementation requirements on the unbuffered I/O functions. This section discusses those requirements, after first presenting general information on standard I/O file pointers and their relationship to unbuffered I/O file descriptors.

Before standard I/O can be performed on a file or device, an unbuffered I/O file descriptor must be assigned to it, and a standard I/O "file pointer" must be assigned to the file descriptor. The assignment of a file pointer and file descriptor can be done dynamically, by calling the standard I/O **fopen()** function. Three file pointers, named **stdin**, **stdout**, and **stderr**, are pre-assigned to file descriptors 0, 1, and 2; these file descriptors in turn are pre-assigned to the console.

When a program calls a standard I/O function, it often must pass a file pointer, which identifies the file or device on which I/O is to be performed. There are a special set of standard I/O functions for accessing **stdin**, **stdout**, and **stderr**: for these, the file pointer isn't passed, since the functions know what file pointer is being accessed.

Supporting the standard I/O fopen() and fclose() functions

The dynamic assignment of a file pointer and file descriptor to a file or device is done by the **fopen()** function. This function selects a file pointer for the file or device and then calls the unbuffered I/O **open()** function, which selects a file descriptor.

If programs call **fopen()**, you must implement the unbuffered I/O **open()** function, and **open()** must return the file descriptor that's associated with the file or device. This requirement (for a functional **open()** when **fopen()** is called) must be met even if file descriptors are pre-assigned to devices; **open()** in this case could be very simple, just searching a table for a device name and returning the associated file descriptor.

Conversely, the use of the standard I/O functions to access those devices that don't first have to be **fopen()**ed (i.e. **stdin**, **stdout**, and **stderr**) places no requirements on **open()**. In particular, if file descriptors are pre-assigned to devices and **open()** simply returns when called, programs can still call the standard I/O functions to access the devices associated with the **stdin**, **stdout**, and **stderr** file pointers.

The standard I/O function **fclose()** calls the unbuffered I/O function **close()**. Thus, if programs call **fclose()**, you must implement a **close()** function. If assignments of devices to file descriptors is hard-coded, **close()** can usually just return the value 0, since nothing special (such as calling the operating system to close an open file or deallocating a file descriptor) needs to be done.

Supporting the standard I/O input and output functions

If programs call any of the standard I/O input functions, you must implement the unbuffered I/O **read()** function. And if they call any of the standard I/O output functions, you must implement the **write()** function.

Supporting the standard I/O fseek() function

If programs will call the standard I/O **fseek()** function, you must implement the unbuffered I/O **lseek()** function, since **fseek()** calls **lseek()**.

Standard I/O and the isatty() function

If programs call any standard I/O functions, you must implement the unbuffered I/O function **isatty()**. The standard I/O functions call this function to decide whether their I/O to a file or device should be buffered or unbuffered.

This use of the word "unbuffered" in describing standard I/O might be a little confusing, since the use of the expression "unbuffered I/O functions" to describe one set of I/O functions implies that the other set, the "standard I/O functions", are buffered. Nevertheless, a standard I/O stream can be either buffered or unbuffered: if buffered, data that's exchanged between user-written functions and the unbuffered I/O functions passes through a buffer; if unbuffered, data doesn't pass through a buffer.

For a given file descriptor, **isatty()** should return non-zero if standard I/O to the device associated with the file descriptor is to be buffered, and zero if it is to be unbuffered.

For example, **isatty()** should probably return non-zero for a file descriptor that's associated with the system console and zero for file descriptors associated with files; it could return either zero or non-zero for other devices, such as printers, depending on your system's requirements.

Unbuffered I/O Names

There are two entry points to an unbuffered I/O function. These entry points are named as follows:

- One entry point has the name that has been used in the above paragraphs. **(open(), close(), read(), write())**
- The other entry point has the same name as the first, but with a prepended underscore. **(_open(), _close(), _read(), _write())**

The functions that do the read work (that is, the ones you must implement) are the ones with the prepended underscore. The code that is a at a name that does not have a prepended underscore is a one line assembly instruction that simply jumps to the corresponding name that has the prepended underscore. For example, the code at **open()** is **jmp _open_**.

The standard I/O routines call the unbuffered I/O routines at the entry point whose name has the prepended underscore. For example, **fopen()** calls **_open()**. Thus, you should follow this naming convention when you implement the unbuffered I/O functions.

The unbuffered I/O routines are named in this fashion to avoid what is called NAME SPACE POLLUTION. That is, the only names that are allowed in an ANSI compatible library, when the library is used by an ANSI conforming program, are those that have a leading underscore.

Unbuffered I/O Return Codes

We've presented most of the factors you should consider when writing your unbuffered I/O functions. In this section we want to list error codes that the functions could return in the global **short errno**.

open() error codes:

ENOENT	File does not exist and O_CREAT wasn't specified.
EEXIST	File exists, and O_CREAT+O_EXCL was specified.
EMFILE	Invalid file descriptor passed to **open()**.

close() error codes:

EBADF	Bad file descriptor passed to **close()**.

creat() error codes:

EMFILE	All file descriptors are in use.

lseek() error codes:

EBADF	Invalid file descriptor
EINVAL	Offset parameter is invalid, or the requested position is before the beginning of the file.

read() error codes:

EBADF	Invalid file descriptor

write() error codes:

EBADF	Invalid file descriptor
EINVAL	Invalid operation; i.e. writing not allowed.

Modifying the sbrk() and brk() Functions

sbrk() and **brk()** provide an elementary means of allocating and deallocating space from a program's heap. **sbrk()** is called by the more sophisticated heap-allocation functions (**malloc()**, etc), and **malloc()** is called by the standard I/O functions; thus, if your programs call **malloc()** or the other high-level heap management functions, or if they call the standard I/O functions, you will need to write an **sbrk()** function.

You probably won't have to modify **sbrk()** or **brk()**, since the most system-dependent code (which defines the boundaries of the heap) is in the startup routine. But if you do, here are some things you should consider:

- A buffer allocated by **sbrk()** should be on a quad-byte boundary (i.e. the address of its first byte should be divisible by four), since words on a 68000 or 68010 must be on an even-byte boundary and since long words on a 68020 can be most efficiently accessed if they're on a quad-word boundary.

- **malloc()** assumes that the heap is a single, contiguous section of memory: when told to allocate a large block of memory, **malloc()** makes repeated calls to **sbrk()** for small blocks of memory, and then attempts to coalesce the small blocks into one large block.

Modifying the exit() and _exit() Functions

exit() and **_exit()** are called to terminate the execution of a program. They are not usually called by ROM-based programs, since such programs usually do not terminate.

They are called, however, by RAM-based programs that are running in an operating system environment, since these programs usually do terminate.

When these functions are needed, you will have to modify **_exit()** , since it must return to the operating system. But you can probably use **exit()** as is, since it closes open files and devices in a system-independent way and then calls **_exit()** .

Modifying the time() and clock() Functions

Some of the functions provided with Aztec C68k/ROM are system independent. However, the **time()** and **clock()** functions are system dependent, and must be specially implemented for your system.

- **time()** is called by the other time functions. They assume that **time()** returns the number of seconds that have elapsed since Jan1, 1970.
- **clock()** is used to determine the amount of time elapsed between two events. **clock()** is not called by any Aztec functions, so you are free to implement any way you want.

Building the libraries

Once you've made modifications to the supplied library functions, you can build your libraries. We've provided makefiles (which give directions to the **make** program) and **lb68** command files that will make this task easier; they can make the following libraries:

c.lib	General purpose functions;
m.lib	Floating point functions ;
m8.lib	68881 functions;

The modules in these three libraries are compiled to use 32-bit **int**s and the small code, small data memory model.

The makefiles can also generate three other variants of each of the above libraries, variants that use different combinations of **int** size (16 or 32 bits) and memory model (small code, small data or large code, large data). The name of one of these variant libraries is derived from the 32-bit **int**, small memory model library by adding **16** to the name if the library uses 16-bit **int**s, and/or **l** if the library uses large memory model

If you followed our recommendations for installing Aztec C68k/ROM, each of the LIB directory's subdirectories contains a makefile that causes **make** to compile and assemble the subdirectory's source files. There is a makefile in the LIB directory that will have **make** first generate each subdirectory's object modules and then make a library.

Before you can generate the libraries, you must do several things:

- In each makefile, modify the rules that define how to convert a C source file to an object module, so that the command that starts the compiler uses the options that correctly define register usage on your system;

- If you've written your own unbuffered I/O modules, you'll probably need to modify the makefile that's in the ROM68 directory;

- In the LIB \INP directory are several files whose extension is **.inp**. Each of these files tells **lb68** how to create a library. For example, **c.inp** is used to create **c.lib**; and **c16l.inp** is used to create **c16l.lib**.

- The environment variable INCL68 must be set to the name of the 'include' directory; that is, to the name of the directory that contains the include files. This is done using the **set** command.

- If you have a RAM disk, you can speed up the library-generation process by defining it using the CCTEMP environment variable. For more information, see the description of CCTEMP in the **Compiler** chapter.

You are now ready to create the libraries. Set the default or current directory to the LIB directory and start **make**, passing to it a code that defines the libraries to make. If you do not specify a code **make** will create all versions of all libraries.

Some of the codes that can be passed to **make** are:

c	Make all versions of **c.lib**
m	Make all versions of **m.lib**
m8	Make all versions of **m8.lib**

To make a specific library, you pass **make** a 3 or 4 letter code:

- The first letter is **c**, **m**, or **m8**, which identifies the library;
- The next letter identifies the **int** size: **s** for 16 bit **int**s, **l** for 32;
- The last letter identifies the memory model: **s** for small code/small data, **l** for large code, large data

For example, typing

```
make cls
```

will make **c.lib**, whose modules use 32-bit **int**s and the small code/small data memory model, and typing

```
make m8sl
```

will make **m8l16.lib**, whose modules use 16-bit **int**s and the large code/large data memory model.

Once started, **make** will activate several other copies of **make**, each of which will compile and assemble the files in one of LIB's subdirectories; it will then start **lb68**, which will make the specified library from the object modules that are in the subdirectories, as directed by the appropriate **.inp** file.

The generated libraries are placed in the **lib\libs** directory.

At times during library generation, there will be two copies of make in memory, and another program. If your system doesn't have enough memory to hold all of these programs (in this case, **make** will abort with the message "EXEC failure"), it may still have enough to hold one copy of **make** and another program. In this case, you can create a batch file of the commands generated by the top-level **make** program, and then execute the batch file, invoke **make** with the **-n** option and redirect its output to a file.

Library Directories

The LIB directory contains the following subdirectories:

Directory	Contents
INP	Files that describe the contents of each library, and that are used by **lb68** when building the libraries;
LIBS	Object module libraries;
MX_IEEE	Manx IEEE math library source and object modules;
M881	68881 math library modules;
MISC	Miscellaneous modules;
STDIO	Standard I/O modules (those that are prototyped in **stdio.h**);
STDLIB	Standard library modules (those that are prototyped in **stdlib.h**);
STRING	String modules;
ROM68	ROM specific routines;
ROM-AMI	Routines for testing C68k/ROM-generated programs on an Amiga;
ROM-MAC	Routines for testing C68k/ROM-generated programs on an Macintosh;
TIME	Time library modules (for routines prototyped in **time.h**).

Chapter 9 - Library Overview

This chapter presents an overview of the functions that are provided with Aztec C. It is divided into the following sections:

I/O	Introduces the I/O system provided in the Aztec C package.
Standard I/O	The I/O functions can be grouped into two sets; this section describes one of them, the standard I/O functions.
Unbuffered I/O	Describes the other set of I/O functions, the unbuffered.
Console I/O	Describes special topics relating to console I/O.
Dynamic Buffer Allocation	Discusses topics related to dynamic memory allocation.
Errors	Presents an overview of error processing.
Header Files	Describes the header files provided with Aztec C.

The unbuffered I/O and console I/O functions are system dependent; the standard I/O functions are system independent, but they call the unbuffered I/O functions. Thus, before you can call any of the I/O functions, the unbuffered I/O functions must be implemented for your system.

For information on implementing the unbuffered I/O functions, see the **Library Customization** chapter.

Overview of I/O

There are two sets of functions for accessing files and devices: the unbuffered I/O functions and the standard I/O functions. The Aztec C standard I/O functions conform to the ANSI standard, while the unbuffered I/O functions behave like those described in chapters 7 and 8 of *The C Programming Language*. The unbuffered I/O functions are so called because, with few exceptions, they transfer information directly between a program and a file or device. By contrast, the standard I/O functions maintain buffers through which data must pass on its journey between a program and a disk file.

The unbuffered I/O functions are used by programs which perform their own blocking and deblocking of disk files. The standard I/O functions are used by programs which need to access files but do not want to be bothered with the details of blocking and deblocking the file records.

The unbuffered and standard I/O functions each have their own overview sections, "Unbuffered I/O" and "Standard I/O". The remainder of this section discusses features which the two sets of functions have in common.

The basic procedure for accessing files and devices is the same for both standard and unbuffered I/O: the device or file must first be "opened", that is, prepared for processing; then I/O operations occur; then the device or file is "closed".

There is a limit on the number of files and devices that can simultaneously be open; the limit on your system is FOPEN_MAX. This macro is defined in **stdio.h** as 20.

Each set of functions has its own functions for performing these operations. For example, each set has its own functions for opening a file or device. Once a file or device has been opened, it can be accessed only by functions in the same set as the function which performed the open, and must be closed by the appropriate function in the same set. There are exceptions to this non-intermingling which are described below.

There are two ways a file or device can be opened: first, the program can explicitly open it by issuing a function call. Second, it can be associated with one of the logical devices standard input, standard output, or standard error, and then opened when the program starts.

Pre-opened Devices and Command Line Arguments

Many versions of Aztec C generate programs that run on systems that have an operating system, such as the IBM PC, Macintosh, and Amiga. The Aztec provided startup code for these systems provide programs with several useful features:

- Three logical devices are pre-opened for the program. They are called standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**); by default, they are associated with the system console.

- **stdin, stdout,** and **stderr** can be redirected by the operator to another device or file.

- The operator can pass information to the program on the line that invokes the program. This information is called COMMAND LINE ARGUMENTS.

For ROM based systems, we do not know the environment in which a program will run, and hence can not provide startup code that provides these features. However, we will describe these features in this section, just in case they are implemented on your system. To find out if they are implemented, contact your system implementor.

Redirecting stdin and stdout

- To redirect **stdin** enter *<file* on the command line that starts the program, where *file* is the name of the device or file to which **stdin** is to be redirected.

- To redirect **stdout**, enter *>file* on the command line, where *file* is the name of the file or device to which **stdout** is to be redirected.

For example, suppose the executable program **cpy** reads standard input and writes it to standard output. Then the following command will read lines from the keyboard and write them to the display

```
cpy
```

The following will read from the keyboard and write it to the file **testfile**

```
cpy > testfile
```

This will copy the file **exmplfil** to the console:

```
cpy < explfil
```

And this will copy **exmplfil** to **testfile**:

```
cpy < exmpfil > testfile
```

Command Line Arguments

Command line arguments can be passed to the user's program via the user's function **main**(*argc, argv*). *argc* is an integer containing the number of arguments plus one; *argv* is a pointer to a an array of character pointers, each of which, except the first, points to a command line argument. On some systems, the first array element points to the command name; on others, it is a null pointer.

For example, if the following command is entered:

```
prog arg1 arg2 arg3
```

the program **prog** will be activated and execution begins at the user's function **main**. The first parameter to **main** is the integer **4**. The second parameter is a pointer to an array of four charac-

ter pointers; on some systems the first array element will point to the string **prog** and on others it will be a null pointer. The second, third, and fourth array elements will be pointers to the strings **arg1**, **arg2**, and **arg3** respectively.

The command line can contain both arguments to be passed to the user's program and I/O redirection specifications. The I/O redirection strings won't be passed to the user's program, and can appear anywhere on the command line after the command name. For example, the standard output of the **prog** program can be redirected to the file **outfile** by any of the following commands; in each case the **argc** and **argv** parameters to the **main** function of **prog** are the same as if the redirection specifier was not present:

```
prog arg1 arg2 arg3 > outfile
prog > outfile arg1 arg2 arg3
prog arg1 > outfile arg2 arg3
```

File I/O

A program can access files both sequentially and randomly, as discussed in the following paragraphs.

Sequential I/O

For sequential access, a program issues any of the various read or write calls. The transfer will begin at the file's current position, and will leave the current position set to the byte following the last byte transferred. A file can be opened for read or write access; in this case, its current position is initially the first byte in the file. A file can also be opened for append access; in this case its current position is initially the end of the file.

On systems which do not keep track of the last character written to a file, it is not always possible to correctly position a file to which data is to be appended. To see if this is a problemon your system, contact your system implementor.

Random I/O

Two functions are provided which allow a program to set the current position of an open file: **fseek()**, for a file opened for standard I/O; and **lseek()**, for a file opened for unbuffered I/O.

A program accesses a file randomly by first modifying the file's current position using one of the **seek** functions. Then the program issues any of the various read and write calls, which sequentially access the file.

A file can be positioned relative to its beginning, current position, or end. Positioning relative to the beginning and current position is always correctly done. For systems which do not keep track of the last character written to a file, positioning relative to the end of a file cannot always be correctly done.

Opening Files

Opening files is somewhat system dependent: the parameters to the **open** functions are the same on the Aztec C packages for all systems, but some system dependencies exist, to conform with the system conventions. For example, the syntax of file names and the areas searched for files differ from system to system.

For information on opening files on your system, contact your system implementor.

Device I/O

Aztec C allows programs to access devices as well as files. Each system has its own names for devices. For the names of devices on your system, contact your system implementor.

Console I/O

Console I/O can be performed in a variety of ways. There's a default mode, and other modes can be selected by calling the function **ioctl()**. We will briefly describe console I/O in this section; for more details, see the Console I/O section of this chapter.

When the console is in default mode, console input is buffered and is read from the keyboard a line at a time. Typed characters are echoed to the screen and the operator can use the standard operating system line editing facilities. A program does not have to read an entire line at a time (although the system software does this when reading keyboard input into it is internal buffer), but at most one line will be returned to the program for a single read request.

The other modes of console I/O allow a program to get characters from the keyboard as they are typed, with or without their being echoed to the display; to disable normal system line editing facilities; and to terminate a read request if a key is not depressed within a certain interval.

Output to the console is always unbuffered: characters go directly from a program to the display. The only choice concerns translation of the newline character; by default, this is translated into a system dependent sequence of characters. This translation can be disabled.

I/O to Other Devices

On most systems, few options are available when writing to devices other than the console. For a discussion of such options, if any, that are available on your system, contact your system implementor.

Mixing Unbuffered and Standard I/O Calls

As mentioned above, a program generally accesses a file or device using functions from one set of functions or the other, but not both.

However, there are functions which facilitate this dual access: if a file or device is opened for standard I/O, the function **fileno()** returns a file descriptor which can be used for unbuffered access to the file or device. If a file or device is open for unbuffered I/O, the function **fdopen()** will prepare it for standard I/O as well.

Care is warranted when accessing devices and files with both standard and unbuffered I/O functions.

Overview of Standard I/O

The standard I/O functions are used by programs to access files and devices. Aztec C functions are totally ANSI compatible.

These functions provide programs with convenient and efficient access to files and devices. When accessing files, the functions buffer the file data; that is, handle the blocking and deblocking of file data. Thus the user's program can concentrate on its own concerns.

Buffering of data to devices when using the standard I/O functions is discussed below.

For programs which perform their own file buffering, another set of functions are provided. These are described in the section "Unbuffered I/O" of this chapter.

Before you can call the standard I/O functions, the unbuffered I/O functions must be implemented for your system.

Opening Files and Devices

Before a program can access a file or device, it must be "opened", and when processing on it is done it must be "closed".

An open device or file is called a STREAM and has associated with it a pointer, called a FILE POINTER, to a structure of type FILE. This identifies the file or device when standard I/O functions are called to access it.

There are two ways for a file or device to be opened for standard I/O: first, the program can explicitly open it, by calling one of the functions **fopen()**, **freopen()**, or **fdopen()**. In this case, the **open()** function returns the file pointer associated with the file or device. **fopen()** just opens the file or device. **freopen()** reopens an open stream to another file or device; it is mainly used to change the file or device associated with one of the logical devices standard output, standard input, or standard error. **fdopen()** opens for standard I/O a file or device already opened for unbuffered I/O.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file pointer is **stdin**, **stdout**, or **stderr**, respectively. These symbols are defined in the header file **stdio.h**.

Closing Streams

A file or device opened for standard I/O can be closed in two ways:

- first, the program can explicitly close it by calling the function **fclose()**.
- when the program terminates, either by falling off the end of the function **main()**, or by calling the function **exit()**, the system will automatically close all open streams.

Letting the system automatically close open streams is error-prone: data written to files using the standard I/O functions is buffered in memory, and a buffer is not written to the file until it is full or the file is closed. Most likely, when a program finishes writing to a file, the file's buffer will be partially full, with this information not having been written to the file. If a program calls **fclose()**, this function will write the partially filled buffer to the file and return an error code if this couldn't be done. If the program lets the system automatically close the file, the program won't know if an error occurred on this last write operation.

Sequential I/O

Files can be accessed sequentially and randomly. For sequential access, simply issue repeated read or write calls; each call transfers data beginning at the current position of the file, and updates the current position to the byte following the last byte transferred. When a file is opened, its current position is set to zero, if opened for read or write access, and to its end if opened for append.

On systems which do not keep track of the last character written to a file, not all files can be correctly positioned for appending data. See the section entitled I/O for details.

Random I/O

The function **fseek()** allows a file to be accessed randomly, by changing its current position. Positioning can be relative to the beginning, current position, or end of the file.

For systems which do not keep track of the last character written to a file, positioning relative to the end of a file cannot always be correctly done. See the I/O overview section for details.

Buffering

When the standard I/O functions are used to access a file, the I/O is buffered. Either a user-specified or dynamically- allocated buffer can be used.

The user's program specifies a buffer to be used for a file by calling the function **setbuf()** or **setvbuf()** after the file has been opened but before the first I/O request to it has been made.

If, when the first I/O request is made to a file, the user hasn't specified the buffer to be used for the file, the system will automatically allocate, by calling **malloc()**, a buffer for it. When the file is closed it is buffer will be freed, by calling **free()**.

Dynamically allocated buffers are obtained from the one region of memory (the heap), whether requested by the standard I/O functions or by the user's program. For more information, see the overview section "Dynamic Buffer Allocation."

The default size of an I/O buffer is given by the manifest constant BUFSIZE in **stdio.h**. Use **setvbuf()** to set a different buffer size.

By default, output to the console using standard I/O functions is unbuffered; all other device I/O using the standard I/O functions is buffered. Console input buffering can be disabled using the **setvbuf()** function; see the overview section "Console I/O" for details.

Errors

There are three fields which may be set when an exceptional condition occurs during stream I/O. Two of the fields are unique to each stream (that is, each stream has its own pair). The other is a global integer.

One of the fields associated with a stream is set if end of file is detected on input from the stream; the other is set if an error occurs during I/O to the stream. Once set for a stream, these flags remain set until the stream is closed or the program calls the **clearerr()** function for the stream. The only exception to the last statement is that when called, **fseek()** will reset the end of file flag for a stream. A program can check the status of the eof and error flags for a stream by calling the functions **feof()** and **ferror()**, respectively.

The other field which may be set is the global integer **errno**. By convention, a system function which returns an error status as its value can also set a code in **errno** which more fully defines the error.

If an error occurs when a stream is being accessed, a standard I/O function returns EOF (-1) as its value, after setting a code in **errno** and setting the stream's error flag.

If end of file is reached on an input stream, a standard I/O function returns EOF after setting the stream's end of file flag.

There are two techniques a program can use for detecting errors during stream I/O. First, the program can check the result of each I/O call. Second, the program can issue I/O calls and only periodically check for errors (for example, check only after all I/O is completed).

On input, a program will generally check the result of each operation.

On output to a file, a program can use either error checking technique; however, periodic checking by calling **ferror()** is more efficient. When characters are written to a file using the standard I/O functions they are placed in a buffer, which is not written to disk until it is full. If the buffer is not full, the function will return good status. It will only return bad status if the buffer was full and an error occurred while writing it to disk. Since the buffer size is 1024 bytes, most write calls will return good status, and hence periodic checking for errors is sufficient and most efficient.

Once a file opened for standard I/O is closed, **ferror()** cannot be used to determine if an error has occurred while writing to it. Hence **ferror()** should be called after all writing to the file is completed but before the file is closed. The file should be explicitly closed by **fclose()**, and its return value checked, rather than letting the system automatically close it, to know positively whether an error has occurred while writing to the file. The reason for this is that when the writing to the file is completed, it is standard I/O buffer will probably be partly full. This buffer will be written to the file when the file is closed, and **fclose()** will return an error status if this final write operation fails.

The Standard I/O Functions

The standard I/O functions can be grouped into two sets: those that can access only the logical devices standard input, standard output, and standard error; and all the rest.

Here are the standard I/O functions that can only access **stdin**, **stdout**, and **stderr**. These are all ASCII functions; that is, they expect to deal with text characters only. Those preceded with an asterisk (*) are non-ANSI functions.

getchar()	Get an ASCII character from **stdin**
gets()	Get a line of ASCII characters from **stdin**
printf()	Format data and send it to **stdout**
***puterr()**	Send a character to **stderr** (obsolete)
putchar()	Send a character to **stdout**
puts()	Send a character string to **stdout**
scanf()	Get a line from **stdin** and convert it
vprintf()	Format data using a variable argument list and send it to **stdout**

Here are the rest of the standard I/O functions:

clearerr()	Clear EOF and error flag for stream
fclose()	Close an open stream
***fdopen()**	Open as a stream a file or device already opened for unbuffered I/O
fclose()	Close an open stream
feof()	Check for end of file on a stream
ferror()	Check for error on a stream
fflush()	Write stream's buffer
fgetc()	Get the character from the input stream
fgetpos()	Get information on file position
fgets()	Get a line of ASCII characters
***fileno()**	Get file descriptor associated with stream
fopen()	Open a file or device
fprintf()	Format data and write it to a stream
fputc()	Write a character to an output stream
fputs()	Send a string of ASCII characters to a stream
fread()	Read binary data
freopen()	Open an open stream to another file or device
fscanf()	Get data and convert it
fseek()	Set current position within a file
fsetpos()	Set file position
ftell()	Get current position
fwrite()	Write binary data
getc()	Get a binary character
***getw()**	Get two binary characters
perror()	Map **errno** to a descriptive string
putc()	Send a binary character
***putw()**	Send two binary characters
setbuf()	Specify buffer for stream
setvbuf()	Specify buffer, buffer size, and buffering method

tmpnam()	Generates a unique name for a file
ungetc()	Push character back into stream
vfprintf()	Format data using a variable argument list and write to a stream

Overview of Unbuffered I/O

The unbuffered I/O functions are used to access files and devices. They are compatible with their UNIX counterparts with the addition that a file may be opened as either text or binary.

As their name implies, a program using these functions, with two exceptions, communicates directly with files and devices; data does not pass through system buffers. Some unbuffered I/O, however, is buffered. When data is transferred to or from a file in blocks smaller than a certain value, it is buffered temporarily. This value differs from system to system, but is always less than or equal to 512 bytes. Also, console input can be buffered, and is, unless specific actions are taken by the user's program.

Programs which use the unbuffered I/O functions to access files generally handle the blocking and deblocking of file data themselves. Programs requiring file access but unwilling to perform the blocking and deblocking can use the standard I/O functions; see the overview section "Standard I/O" for more information.

Here are the unbuffered I/O functions:

close()	Conclude the I/O on an open file or device
creat()	Create a file and open it
ioctl()	Change console I/O mode
isatty()	Is an open file or device the console?
lseek()	Change the current position of an open file
open()	Prepare a file or device for unbuffered I/O
read()	Read data from an open file or device
remove()	Delete a file
rename()	Rename a file
unlink()	Delete a file
write()	Write data to an open file or device

Before a program can access a file or device, it must be "opened", and when processing on it is done, it must be "closed".

An open file or device has an integer known as a **FILE DESCRIPTOR** associated with it; this identifies the file or device when it is accessed.

There are two ways for a file or device to be opened for unbuffered I/O. First, it can explicitly open it, by calling the function **open()**. In this case, **open()** returns the file descriptor to be used when accessing the file or device.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file descriptor is the integer value 0, 1, or 2, respectively. See the section entitled I/O for more information on this.

An open file or device is closed by calling the function **close()**. When a program ends, any devices or files still opened for unbuffered I/O will be closed.

If an error occurs during an unbuffered I/O operation, the function returns -1 as its value and sets a code in the global integer **errno**. For more information on error handling, see the section "Errors".

The unbuffered I/O routines are system dependent and must be specially written for your system.

The remainder of this section discusses unbuffered I/O to files and devices.

File I/O

Programs call the functions **read()** and **write()** to access a file; the transfer begins at the current position of the file and proceeds until the number of characters specified by the program have been transferred.

The current position of a file can be manipulated in various ways by a program, allowing both sequential and random acccess to the file. For sequential access, a program simply issues consecutive I/O requests. After each operation, the current position of the file is set to the character following the last one accessed.

The function **lseek()** provides random access to a file by setting the current position to a specified character location.

lseek() allows the current position of a file to be set relative to the end of a file. For systems which do not keep track of the last character written to a file, such positioning cannot always be correctly done. For more information, see the section entitled I/O.

open() provides a mode, O_APPEND, which causes the file being opened to be positioned at its end. This mode is supported on UNIX Systems 3 and 5, but not UNIX version 7. As with **lseek()**, the positioning may not be correct for systems which do not keep track of the last character written to a file.

open() also provides for opening files as text by using the O_TEXT mode. The default mode is binary. The unbuffered I/O routines can access files in text or binary mode.

- When accessing a file in text mode, the unbuffered I/O routines perform system dependent end-of-line translations. For example, **write()** might translate '\n' to '\r\n' and **read()** might translate '\r\n' to '\n'.

- When accessing a file in binary mode, no translations are done. The mode in which a file is to be accessed is specified when the file is opened; by including one of the following in the open functions mode parameter:

O_TEXT	access file in text mode
O_BINARY	access file in binary mode

Device I/O

Unbuffered I/O to the Console

There are several options available when accessing the console, which are discussed in detail in the Console I/O sections of this chapter and of the system-dependent appendix to this chapter. Here we just want to briefly discuss the line or character modes of console I/O as they relate to the unbuffered I/O functions.

Console input can be either line or character oriented. With line-oriented input, characters are read from the console into an internal buffer a line at a time, and returned to the program from this buffer. Line buffering of console input is available even when using the so-called "unbuffered" I/O functions.

With character oriented input, characters are read and returned to the program when they are typed: no buffering of console input occurs.

Unbuffered I/O to Non-Console Devices

Unbuffered I/O to devices other than the console is truly unbuffered.

Overview of Console I/O

The system dependent function **ioctl()** gives a program control over several options relating to console I/O. However, since **ioctl()** is system dependent, it must be implemented for your system before you can use it.

Some of the choices for console I/O that **ioctl()** gives you are these:

- Console I/O can be either line or character oriented.
- The echoing of typed characters can be enabled or disabled, using the ECHO option.
- End-of-line translation can be enabled and disabled, using the CRMOD option.
- Other choices may be available. For details contact your system implementor.

All the console I/O options have default settings, which allow a program to easily access the console without having to set the options itself. In the default mode, console I/O is line-oriented, with ECHO and CRMOD enabled.

Console I/O behaves the same on all systems when the console options have their default settings. However, the behavior of console I/O differs from system to system when the options are changed from their default values. Thus, a program requiring machine independence should either use the console in its default mode or be careful how it sets the console options. In the paragraphs below, we will try to point out system dependencies.

Line-Oriented Input

With line-oriented input, a program issuing a read request to the console will wait until an entire line has been typed. On some systems a non-UNIX option (NODELAY) is available that will prevent this waiting.

The program need not read an entire line at once; the line will be internally buffered, and characters returned to the program from the buffer, as requested. When the program issues a read request to the console and the buffer is empty, the program will wait until an entire new line has been typed and stored in the internal buffer Again, on some systems programs can disable this wait by setting the NODELAY option.

A single unbuffered read operation can return at most one line.

On most systems, selecting line-oriented console input forces the ECHO option to be enabled. On such systems the program still has control over the CRMOD option. To find out if, on your system, line-oriented mode always has ECHO enabled, consult with your system implementor.

Character-oriented input

The basic idea of character-oriented console input is that a program can read characters from the console without having to wait for an entire line to be entered.

The behavior of character-oriented console input differs from system to system, so programs requiring both machine independence and character-oriented console input have to be careful in their use of the console. However, it is possible to write such programs, although they may not be able to take full advantage of the console I/O features available for a particular system.

There are two varieties of character-oriented console input, named CBREAK and RAW. Their primary difference is that with the console in CBREAK mode, a program still has control over the other console options, whereas with the console in RAW mode it does not. In RAW mode, all other console options are reset: ECHO and CRMOD are disabled.

Thus, to some extent RAW mode is simply an abbreviation for CBREAK on, all other options off.

Writing System-Independent Programs

To write system-independent programs that access the console in character-oriented input mode, the console should be set in RAW mode, and the program should read only a single character at a time from the console. All the non-UNIX options that are supported by some systems should be reset.

The standard I/O functions all read just one character at a time from the console, even when the calling program requests several characters. Thus, programs requiring system independence and character-oriented input can read the console using the standard I/O functions.

Some systems require a program that wants to set console option to first call **ioctl()** to fetch the current console options, then modify them as desired, and finally call **ioctl()** to reset the new console options. The systems that do not require this do not care if a program first fetches the console options and then modifies them. Thus, a program requiring system-independence and console I/O options other than the default should fetch the current console options before modifying them.

Using ioctl()

A program that calls **ioctl()** should include the files **fcntl.h** and **sgtty.h**. **fcntl.h** contains a prototype for **ioctl()**, and **sgtty.h** defines symbolic constants and structures that are used when calling **ioctl()**. A call to **ioctl()** has the following form:

ioctl(*fd, code, arg*)

where the arguments are as follows:

- *fd* is a file descriptor associated with the console. On UNIX, this parameter defines the file descriptor associated with the device to which the **ioctl()** call applies.

- *code* defines the action to be performed by **ioctl()**. It can have the following values:

TIOCGETP	Fetch the console parameters and store them in the structure pointed at by *arg*.
TIOCSETP	Set the console parameters according to the structure pointed at by *arg*.
TIOCSETN	Equivalent to TIOCSETP.
TIOCNTLC	Aborts program if system dependent abort keys have been entered.

- *arg* points to a structure named **sgttyb** that contains the following fields:

```
int sg_flags;
char sg_erase;
char sg_kill;
```

The order of these fields is system-dependent.

The **sg_flags** field is supported by all systems, while the other fields are not supported by some systems.

To set console options, a program should fetch the current state of the **sgtty** fields, using **ioctl**'s TIOCGETP option. Then it should modify the fields to the appropriate values and call **ioctl()** again, using **ioctl**'s TIOCSETP option.

The sgtty Fields

The sg_flags Field

sg_flags contains the following UNIX-compatible flags:

RAW	Set RAW mode (turns off other options). By default, RAW is disabled.
CBREAK	Return each character as soon as typed. By default, CBREAK is disabled.
ECHO	Echo input characters to the display. By default, ECHO is enabled.
CRMOD	Perform end-of-line translations. By default, CRMOD is enabled.

On some systems, other flags are contained in **sg_flags**. To find out if your system supports other flags, consult your system implementor.

More than one flag can be specified in a single call to **ioctl()**; the values are simply 'or'ed together. If the RAW option is selected, none of the other options have any effect.

When the console I/O options are set and RAW and CBREAK are reset, the console is set in line-oriented input mode.

Examples

Console Input Using Default Mode

The following program copies characters from **stdin** to **stdout**. The console is in default mode, and assuming these streams haven't been redirected by the operator, the program will read from the keyboard and write to the display. In this mode, the operator can use the operating system's line editing facilities, such as backspace, and characters entered on the keyboard will be echoed to the display. The characters entered won't be returned to the program until the operator depresses carriage return.

```
#include <stdio.h>
main()
{
   int c;
   while ((c = getchar()) != EOF)
         putchar(c);
}
```

Console Input - RAW Mode

In this example, a program opens the console for standard I/O, sets the console in RAW mode, and goes into a loop, waiting for characters to be read from the console and then processing them. The characters typed by the operator are not displayed unless the program itself displays them. The input request won't terminate until a character is received. This example assumes that the

console is named con:; on systems for which this is not the case, just substitute the appropriate name.

```
#include <stdio.h>
#include <sgtty.h>
main()
{
   int c;
   FILE *fp;
   struct sgttyb stty;
   if ((fp = fopen("con:", "r") == NULL){
         printf("cannot open the console\n");
         exit();
   }
   ioctl(fileno(fp),TIOCGETP, &stty);
   stty.sg_flags |= RAW;
   ioctl(fileno(fp), TIOCSETP, &stty);
   for (;;){
         c = getc(fp);
         . . .
   }
}
```

Console Input - Console In CBREAK + ECHO Mode

This example modifies the previous program so that characters read from the console are automatically echoed to the display. The program accesses the console via the standard input device. It uses the function **isatty()** to verify that **stdin** is associated with the console; if it is not, the program reopens stdin to the console using the function **freopen()**. Again, the console is assumed to be named con:.

```
#include <stdio.h>
#include <sgtty.h>
main()
{
   int c;
   struct sgttyb stty;
   if (!isatty(stdin))
         freopen("con:", "r", stdin);
   ioctl(0, TIOCGETP, &stty);
   stty.sg_flags |= CBREAK | ECHO;
   ioctl(0, TIOCSETP, &stty);
   for (;;){
         c = getchar();
         . . .
   }
}
```

Overview of Dynamic Buffer Allocation

Several functions are provided for the dynamic allocation and deallocation of buffers from a section of memory called the HEAP. They are:

malloc()	Allocates a buffer
calloc()	Allocates a buffer and initializes it to zeroes
realloc()	Allocates more space to a previously allocated buffer
free()	Releases an allocated buffer for reuse

In addition the UNIX-compatible functions **sbrk()** and **brk()** are provided that provide a more elementary means to allocate heap space. The **malloc()**-type functions call **sbrk()** to get heap space, which they then manage.

Dynamic Allocation of Standard I/O Buffers

Buffers used for standard I/O are dynamically allocated from the heap unless specific actions are taken by the user's program. Standard I/O calls to dynamically allocate and deallocate buffers can be interspersed with those of the user's program.

Programs which perform standard I/O and which must have absolute control of the heap can explicitly define the buffers to be used by a standard I/O stream.

Overview of Error Processing

This section discusses error processing which relates to the global integer **errno**. This variable is modified by the standard I/O, unbuffered I/O, and scientific (eg, **sin()**, **sqrt()**) functions as part of their error processing.

When a standard I/O, unbuffered I/O, or scientific function detects an error, it sets a code in **errno** which describes the error. If no error occurs, the scientific functions do not modify **errno**. If no error occurs, the I/O functions may or may not modify **errno**.

Also, when an error occurs,

- A standard I/O function returns -1 and sets an error flag for the stream on which the error occurred;
- An unbuffered I/O function returns -1;
- A math function returns a value that depends on the type of error that occurred. For example, when overflow occurs a huge numebr is returned. This number is given the name HUGE_VAL in **math.h**.

When performing scientific calculations, a program can check **errno** for errors as each function is called. Alternatively, since **errno** is modified only when an error occurs, **errno** can be checked only after a sequence of operations; if it is non-zero, then an error has occurred at some point in the sequence. This latter technique can only be used when no I/O operations occur during the sequence of scientific function calls.

Since **errno** may be modified by an I/O function even if an error didn't occur, a program cannot perform a sequence of I/O operations and then check **errno** afterwards to detect an error. Programs performing unbuffered I/O must check the result of each I/O call for an error.

Programs performing standard I/O operations cannot, following a sequence of standard I/O calls, check **errno** to see if an error occurred. However, associated with each open stream is an error flag. This flag is set when an error occurs on the stream and remains set until the stream is closed or the flag is explicitly reset. Thus a program can perform a sequence of standard I/O operations on a stream and then check the stream's error flag. For more details, see the standard I/O overview section.

The following table lists the system-independent values which may be placed in **errno**. These symbolic values are defined in the file **errno.h**. Other, system-dependent, values may also be set in **errno** following an I/O operation; these are error codes returned by the operating system. System dependent error codes are described in the operating system manual for a particular system.

The system-independent error codes and their meanings are:

Error Code	Meaning
ENOENT	File does not exist
E2BIG	Not used
EBADF	Bad file descriptor - file is not open or improper operation requested
ENOMEM	Insufficient memory for requested operation
EEXIST	File already exists on creat request
EINVAL	Invalid argument
ENFILE	Exceeded maximum number of open files
EMFILE	Exceeded maximum number of file descriptors
ENOTTY	ioctl() attempted on non-console
EACCES	Invalid access request
ERANGE	Math function value cannot be computed
EDOM	Invalid argument to math function
EIO	I/O error (usually physical)
ENOSPC	No space left on device
EROFS	Read-only file system
EXDEV	Cross-device rename
EAGAIN	Nothing to read

ANSI Header Files

Header files may be included in any order. They may be included more than once in a given scope, without any adverse effects. The header files include:

assert.h

Includes the **assert()** macro which is used for program debugging.

ctype.h

Contains several function declarations and macros useful for testing and mapping characters.

fcntl.h

Contains prototypes for the system dependent functions. Also **#defines** the symbolic values that can be passed to **open()**.

float.h

Contains macros that define various floating point sizes and limits.

limits.h

Contains macros that define the numerical limits for various types.

locale.h

Contains macros and function prototypes supporting the "C" locale necessary for C translation.

math.h

Contains prototypes for the floating point functions.

setjmp.h

Contains prototypes, typedefs, and #defines that are used when calling the setjmp() and longjmp() functions.

signal.h

Contains prototypes for the signal() and raise() functions.

stdarg.h

Contains macros for advancing through a variable length argument list.

stddef.h

Contains miscellaneous typedefs and #defines.

stdio.h

Contains prototypes, typedefs, and #defines that are used when calling the standard I/O functions.

stdlib.h

Contains prototypes, typedefs, and #defines that are used when calling the string conversion, memory management, and other functions.

string.h

Contains prototypes, typedefs, and #defines that are used when calling the str... and mem... functions.

time.h

Contains prototypes, typedefs, and #defines that are used when calling the time related functions.

LIBRARY FUNCTIONS 10

Chapter 10 - Library Functions

This chapter describes the functions that are provided with Aztec C68k/ROM. These functions fall into the following three categories:

- SYSTEM DEPENDENT FUNCTIONS: Before you can use any of these low-level functions you will have to write them! The **Library Customization** chapter discusses these functions.

- SYSTEM INDEPENDENT FUNCTIONS WITH UNDERLYING DEPENDENCIES: These functions directly or indirectly call a system dependent function. For example, the system-independent function **printf()** indirectly calls the system dependent function **_write()**. Before you use these system independent functions you will of course have to write the system dependent functions that they use.

- SYSTEM INDEPENDENT FUNCTIONS: This class of functions are system independent and do not directly or indirectly call any systemdependent functions. You can freely call any of these functions.

If a function is directly or indirectly system dependent, the **NOTES** section in the description of the function will say so.

Description Format

Introductory Information

To help you access the information more easily and efficiently:

- Each function description begins on a new page.
- The function descriptions are listed alphabetically.
- If a function contains any direct or indirect system dependencies, it will be documented in the "Notes" section of the that specific function description, otherwise you may assume that the function is totally systemindependent.

For simplicity, those functions that begin with a leading underscore (such as **_abort()**) are placed alphabetically as if the underscore is not present.

Each library function description begins with the name and a brief definition of the function, followed by:

- a list of **FUNCTIONS**
- the **TYPE** of function
- the function's **COMPATIBILITY**
- the **LIBRARY** in which the function is contained.

Declaration

The **DECLARATION** of the function follows. This is expressed as a "prototype" statement, with all variable information denoted by italics. The declaration indicates the type of arguments that the function requires, and the values it returns. Unless otherwise noted, the function's declaration is contained in the header file specified for that function.

For example, the function **atof()** converts ASCII strings into double precision numbers. It is listed in the declaration as

```
#include <stdlib.h>
double atof(const char *cp);
```

This means that **atof()** returns a value of type **double** and requires as an argument a pointer to a character string. Since **atof()** returns a noninteger value, it must be declared prior to the use of the function.

The notation **#include <stdlib.h>** at the beginning of the declaration indicates that such a statement should appear at the beginning of any program calling **atof()**.

Other Information

A **DESCRIPTION** of the library function follows. Other information may include:

- A **DIAGNOSTICS** section that describes the error codes that the function may return.
- **EXAMPLES** on use of the function.
- A **SEE ALSO** section that lists other relevant functions.
- A **NOTES** section which will give any special system dependency information as well as any other pertinent information that might be helpful in using this function in producing embedded code.

You should also refer to the **Library Overview** chapter for further information about many of the functions discussed in this chapter.

Function List

The library functions described in this chapter include:

abort()	terminate program abnormally
abs()	return the absolute value of an integer
acos()	return the arc cosine of a double value
asctime()	translate a time value into an ASCII string
asin()	return the arc sine of a double value
assert()	verify program assertion
atan()	return the arc tangent of a double value
atan2()	return the arc tangent of a double value *y/x*
atexit()	function to be called at program termination
atof()	convert ASCII string to a double number
atoi()	convert an ASCII string to a signed integer
atol()	convert an ASCII string to a signed long value
brk()	allocates and deallocates space from the heap
bsearch()	search array for matching object
calloc()	allocate space for an array of objects
ceil()	compute smallest integer not less than *x*
clearerr()	clear end of file and error conditions in a stream
clock()	determine time intervals
close()	close a device or file
cos()	return the cosine of a double value
cosh()	return the hyperbolic cosine of a double value
cotan()	return the cotangent of a double value
creat()	create a new file
ctime()	convert a time value to an ASCII string
ctop()	convert string from C to Pascal format
difftime()	compute the difference between two times
div()	compute quotient and remainder
_exit()	
exit()	terminate calling program
exp()	compute the exponential function *ex*
fabs()	return the absolute value of a given number
fclose()	close a buffered I/O stream
fdopen()	open a file or device previously opened
feof()	test for end-of-file in a standard I/O stream
ferror()	test for an error in a standard I/O stream
fflush()	flush an I/O stream
fgetc()	return the next available character
fgetpos()	save the current file position for a stream
fgets()	get a string of characters from a stream
_filbuf()	get and return next character

fileno()	return file descriptor associated with a stream
_fileopen()	open file and associate specified stream with it
floor()	return largest value not greater than input
_flsbuf()	flush specified open stream
fmod()	return the remainder of the double value *x/y*
fopen()	open a file or device for standard I/O access
format()	write formatted data
_format()	write formatted ASCII data to specified stream
fprintf()	write formatted data to an I/O stream
fputc()	write a character to an I/O stream
fputs()	write a string to an I/O stream
fread()	read from a specified standard I/O stream
free()	deallocate a memory block
freopen()	reopen a stream with a new device
frexp()	decompose a floating point number
fscanf()	perform formatted input conversion
fseek()	reposition current location within a stream
fsetpos()	set the correct file position for a stream
ftell()	return the current file position within a stream
ftoa()	convert floating point number to an ASCII string
fwrite()	write to a specified standard I/O stream
_getbuf()	associate a buffer with a stream
getc()	return next available character from a stream
getchar()	return the next available character from stdin
getenv()	get value of environment variable
_getiob()	return next available stdio stream
gets()	get a string of characters from stdin
getw()	read a word from input stream
gmtime()	convert date and time
index()	find the first occurrence of a character in a string
ioctl()	determine and set console mode
is...()	character classification functions
isatty()	determine if device is interactive
labs()	return the absolute value of a signed long
ldexp()	multiply a float by an integral power of 2
ldiv()	compute quotient and remainder of two longs
localeconv()	set components of object for formatting
localtime()	convert a time value relative to local time
log()	compute the natural logarithm of a number
log10()	compute the logarithm of a number to base 10
longjmp()	execute a non-local goto
lseek()	change current position within file
malloc()	allocate a block of system memory
mblen()	determine size of multibyte character
mbstowcs()	convert sequence of multibyte characters

mbtowc()	determine size of multibyte character
memccpy()	copy characters from source to destination
memchr()	find a character within an object
memcmp()	compare two blocks of memory *n* bytes long
memcpy()	copy a block of bytes from one object to another
memmove()	copy a block of memory
memset()	set a block of memory to a specified value
mktime()	convert a time value between formats
modf()	break a floating point value into parts
movmem()	copy a memory block
open()	open device or file for unbuffered I/O
perror()	print a system error message
peek..(),poke..()	get and set bytes in memory
pow()	compute *x* to the *yth* power
printf()	formatted output function
ptoc()	convert string from Pascal to C format
putc()	write a character to an I/O stream
putchar()	write a character to the stdout stream
puts()	write a string to **stdout**
putw()	calls **putc()** to output word to the stream
qsort()	sort an array of records in memory
raise()	send signal to executing program
ran()	generate floating point random numbers
rand()	return a pseudo-random integer
randl()	return a random number
read()	read from a device or file using unbuffered I/O
realloc()	re-allocate memory block to a different size
remove()	delete a file
rename()	rename a disk file
rewind()	reposition a stream's position indicator
rindex()	find last occurrence of a character in a string
sbrk()	increment a pointer by *size* bytes
_scan()	convert text characters from input stream
scanf()	formatted input conversion on stdin stream
setbuf()	associate an I/O stream with a specific buffer
setjmp()	set up for a non-local **goto**
setlocale()	select appropriate portion of program's locale
setmem()	copy value of char into object
setvbuf()	associate an I/O stream with a specific buffer
signal()	define how to handle a signal
sin()	return the sine of a double value
sinh()	return the hyperbolic sine of a double value
sprintf()	write formatted data to a buffer
sqrt()	compute non-negative square root of a value

sran()	set the random number seed for ran
srand()	initialize the seed value used by rand
sscanf()	perform formatted input conversion on buffer
_stkchk	performs stack depth checking
strcat()	concatenate two strings together
strchr()	search for first occurrence of stringcharacter
strcmp()	compare two strings
strcoll()	compare two strings using the current locale
strcpy()	copy one string to another
strcspn()	return the index of specified string
strdup()	copy the string pointed to
strerror()	map error number to error message
strftime()	place characters into array pointed to
strlen()	return the length of a string
strncat()	concatenate two strings together
strncmp()	compare two strings, up to *max* characters.
strncpy()	copy characters from one string to another
strpbrk()	return pointer to first character in a string
strrchr()	search for occurrence of character in string
strspn()	return index of the first character in a string
strstr()	return a pointer of first occurrence of a string
strtod()	convert a string to a double
strtok()	tokenize a string
strtol()	convert a string to a long
strtold()	convert a string to a long double
strtoul()	convert a string to unsigned long integer
strxfrm()	transform a string to match the current locale
swapmem()	swap characters between specified objects
system()	make a call to underlying operating system
tan()	return the tangent of a double value
tanh()	return the hyperbolic tangent of a double value
time()	return the time of day
tmpfile()	create a temporary file
tmpnam()	create a name for a temporary file
tolower()	convert a character to lowercase
toupper()	convert a character to uppercase
ungetc()	push a character back into input stream
unlink()	erase file
va_...()	variable argument access
vfprintf()	write formatted ASCII data to a stream
vprintf()	write formatted ASCII data to stdout
vsprintf()	write formatted ASCII data to a buffer
wcstombs()	convert a sequence of multibyte characters
wctomb()	determine the number of bytes in a multibyte character
write()	write to a file or device using unbuffered I/O

abort() terminate program abnormally

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
void abort(void);
```

DESCRIPTION

abort() causes abnormal program termination to occur, unless the signal SIGABRT is caught and the signal handler does not return. **abort()** calls **raise()** using SIGABRT and, by default, this in turn calls **_exit()**.

NOTES

abort() indirectly calls the system dependent function **_exit()**. Thus, before using **abort()** you must implement **_exit()**.

SEE ALSO

_abort(), exit(), _exit(), raise()

abs() return the absolute value of an integer

TYPE Integer Math
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int abs (int x);
```

DESCRIPTION

abs() computes and returns the absolute value of the signed integer *x*. The largest possible negative integer is returned as itself, since the largest negative integer cannot be represented positively in the size of an integer.

SEE ALSO

labs(), fabs()

acos() return the arc cosine of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double acos (double x);
```

DESCRIPTION

acos() returns the arc cosine of *x*. *x* should be specified in radians. If *x* is outside of the range -1 to 1, then **acos()** will return 0 and set **errno** equal to EDOM.

SEE ALSO

asin(), cos(), sin()

asctime() translate a time value into an ASCII string

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <time.h>
char *asctime (const struct tm *timeptr);
```

DESCRIPTION

asctime() creates an ASCII text string corresponding to the time contained in *timeptr*. The general format of the resulting string is

```
Mon Apr 9 08:29:00 1968\n\o
```

A pointer to the text string is returned by **asctime()**.

SEE ALSO

clock(), ctime(), difftime(), localtime(), time()

asin() return the arc sine of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double asin(double x);
```

DESCRIPTION

asin() returns the arc sine of *x*. *x* should be specified in radians. If *x* is outside of the range -1 to 1, then **asin()** will return 0 and set **errno** equal to EDOM.

SEE ALSO

acos(), cos(), sin()

assert() verify program assertion

TYPE Program Diagnostics Macro
COMPATIBILITY ANSI
LIBRARY **assert.h**

DECLARATION

#include <assert.h>
void assert (int *expression*);

DESCRIPTION

The **assert()** macro is useful for putting diagnostic messages in a program. **assert()** determines whether *expression* is true or false. If *expression* evaluates to false, the message

```
Assertion failed: expr, file ffff, line nnnn
```

is printed to **stderr**, where *ffff* is the name of the source file and *nnnn* is the line number where the assertion failed. Then **abort()** is called to terminate the program.

To prevent assertion statements from being compiled in a program, compile the program with the option **-dNDEBUG**, or place the statement **#define NDEBUG** ahead of the statement **#include assert.h**.

NOTES

assert() indirectly call **_write()** and **_exit()**, which are system dependent. Thus before you call assert, you must implement **_write()** and **_exit()**.

SEE ALSO

abort()

atan() return the arc tangent of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double atan (double x);
```

DESCRIPTION

atan() returns the arc tangent of *x*. *x* should be specified in radians.

SEE ALSO

atan2(), tan()

atan2() return the arc tangent of a double value *y/x*

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double atan2 (double y, double x);
```

DESCRIPTION

atan2() returns the arc tangent of the ratio of the input values (y/x) in the range of -pi to pi. **atan2()** will use the signs of the input values to determine the correct quadrant of the return value. If both x and y are 0, **atan2()** returns 0 and sets **errno** equal to EDOM.

SEE ALSO

atan(), tan()

atexit() function to be called at program termination

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

DESCRIPTION

atexit() is used to indicate that the function *func* should be called at normal program termination. When a program returns from **main()**, or the **exit()** function is called, **atexit()** insures that *func* is called.

Up to 32 functions may be indicated as **exit()** functions with **atexit()**. **atexit()** will return a non-zero value if the function cannot be registered as an **exit()** function; otherwise, it returns 0.

SEE ALSO

exit()

atof() convert ASCII string to a double number

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <stdlib.h>
double atof(const char *cp);
```

DESCRIPTION

atof() converts a string of text characters pointed at by the argument *cp* to a **double**. The string may contain leading blanks and tabs, which it skips, followed by an optional sign (+ or -), then a number containing an optional decimal point (.), then an optional "E" or "e", followed by an optionally signed integer to denote scientific notation. Any characters beyond this are ignored.

EXAMPLE

```
#include <stdlib.h>
main()
{
    double val;
    char *cp;
    cp = "10.6789";
    val = atof (cp);
          printf (" val = %e\n",val);
}
```

SEE ALSO

atoi(), atol(), ftoa(),strtod()

atoi() convert an ASCII string to a signed integer

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int atoi (const char *cp);
```

DESCRIPTION

atoi() converts a string of text characters pointed to by the argument *cp* into a signed integer value, which it returns. The format of the string pointed at by *cp* should contain three components: optional leading blanks or tabs, followed by an optional "+" or "-", followed by an integer. Any numbers following the integer will be ignored, although the string should be null-delimited.

EXAMPLE

```
#include <stdlib.h>
main()
{
    int i;
    char *cp = " 567";
    i = atoi (cp);
    printf ("I = %d\n",i);
}
```

SEE ALSO

atof(), atol(), ftoa(),strtol()

atol() convert an ASCII string to a signed long value

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <stdlib.h>
long int atol (const char *cp);

DESCRIPTION

atol() converts the string of characters pointed to by the argument *cp* to a signed *long*, which is then returned by **atol()**. The string may contain optional leading blanks, followed by an optional "+" or"-", followed by a string of digits. Anything beyond the string of digits is ignored. The string should also be null delimited.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    long val;
    char *cp = " 79832";
    val = atol (cp);
    printf ("val = %ld\n",val);
}
```

SEE ALSO

atof(), atoi(), ftoa(),strtol()

brk() set heap space 'high water' mark

TYPE Memory Allocation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

void *brk (void * *ptr*)

DESCRIPTION

brk() provides an elementary means of deallocating space from the heap. More sophisticated buffer management schemes can be built using this function; for example, the standard functions **malloc()**, **free()**, etc. call **sbrk()** to get heap space, which they then manage for the calling functions.; **sbrk()** in turn calls **brk()**.

brk() sets its internal variable that points at the current top of allocated heap space to *ptr*.

If successful, **brk()** returns 0.

If unsuccessful, **brk()** sets ENOMEM in the global integer **errno** and returns -1.

SEE ALSO

The standard I/O functions usually call **malloc()** and **free()** to allocate and release buffers for use by I/O streams. This is discussed in the Standard I/O section of the Library Functions Overview.

Your program can safely mix calls to the malloc functions, standard I/O calls, and calls to **sbrk()** and **brk()**, as long as the your calls to **sbrk()** and **brk()** don't decrement the heap pointer. Mixing **sbrk()** and **brk()** calls that decrement the heap pointer with calls to the malloc functions and/or the standard I/O functions is dangerous and probably shouldn't be done by normal programs.

bsearch() search array for matching object

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <stdlib.h>
void *bsearch(const void *_key_, const void *_base_, size_t _nmemb_, size_t _size_, int (*_compar_)(const void *, const void *));

DESCRIPTION

bsearch() searches an array of *nmemb* objects, the initial member of which is pointed to by *base*, for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*.

The contents of the array shall be in ascending sorted order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the key object and to an array member, in that order. The function shall return an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array member.

bsearch() returns a pointer to a matching member of the array, or a NULL pointer if no match is found. If two members compare as equal, the member that is matched is unspecified.

SEE ALSO

qsort()

calloc() allocate space for an array of objects

TYPE Memory Allocation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
void *calloc (size_t nmemb, size_t size);
```

DESCRIPTION

calloc() allocates system memory for an array of *nmemb* objects, each *size* bytes long fromthe heap, in a manner similar to the **malloc()** function. The total size of the block will be *size* * *nmemb* bytes long, and each byte within the block is initialized to 0. If **calloc()** is successful, it returns a pointer to the allocated block.

DIAGNOSTICS

If **calloc()** cannot allocate a block large enough to hold *nmemb* elements each *size* bytes long, it will return a NULL pointer. Otherwise, a pointer to the requested block is returned.

SEE ALSO

malloc(), realloc(), free()

ceil() compute smallest integer not less than x

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double ceil (double x);
```

DESCRIPTION

ceil() returns the smallest integral number not less than its input, *x*. The return value is expressed as a **double**. For example, **ceil()** returns 6.0 for an input of 5.3, and -5.0 for an input of -5.3.

SEE ALSO

fabs(), floor(), fmod()

clearerr() clear end of file and error conditions in a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY **c.lib**

DECLARATION

```
#include <stdio.h>
void clearerr (FILE *stream);
```

DESCRIPTION

clearerr() clears both the end-of-file and error condition codes associated with the specified stream. If an error or end-of-file condition occurs on a stream and **clearerr()** is not called, the error condition will remain set until the stream is closed.

SEE ALSO

feof(), ferror(), perror()

clock() determine time intervals

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <time.h>
clock_t clock (void);
```

DESCRIPTION

clock() is used to determine the time interval between two events. The value returned by **clock()** should be divided by the macro **CLF_TCK** to determine the time in seconds.

If the processor time used is not available or its value cannot be represented, **clock()** returns the value (**clock_t**) -1.

NOTES

clock() is system dependent, and must be specially written for your system.

SEE ALSO

difftime(), mktime(), time()

close() close a device or file

TYPE UNIX I/O
COMPATIBILITY Aztec/UNIX
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
int close (int fd);
int _close (int fd);
```

DESCRIPTION

close() closes a device or disk file which has been opened for unbuffered I/O.

The parameter *fd* is the file descriptor associated with the file or device. If the device or file was explicitly opened by the program by calling **open()** or **creat()**, *fd* is the file descriptor returned by **open()** or **creat()**.

close() returns 0 as its value if successful.

_close() performs the same as **close()** and is provided for internal library use in case the user defines a version of **close()** that overrides the library version.

DIAGNOSTICS

If **close()** fails, it returns -1 and sets an error code in the global integer **errno**.

NOTES

_close() and **close()** are system dependent, and must be specially written for your system.

cos() return the cosine of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double cos (double x);
```

DESCRIPTION

cos() returns the cosine of *x*. *x* should be specified in radians.

If the absolute value of *x* is too large, **cos()** will set the symbolic value ERANGE in the global int **errno** and return 0.

SEE ALSO

acos(), asin(), sin()

cosh() return the hyperbolic cosine of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double cosh (double x);
```

DESCRIPTION

cosh() returns the hyperbolic cosine of *x*.

cosh() will return HUGE_VAL and set **errno** equal to ERANGE if *x* is greater than LOGHUGE.

DIAGNOSTICS

The symbolic values are defined in **math.h**.

SEE ALSO

sinh()

cotan() return the cotangent of a double value

TYPE Float Math
COMPATIBILITY Aztec
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double cotan (double x);
```

DESCRIPTION

cotan() returns the cotangent of *x*. *x* should be specified in radians. **cotan()** is not specified by ANSI and may not be portable to other compilers/architectures.

DIAGNOSTICS

Error Codes:

condition	return value	errno
|x|<TINY_VAL	HUGE_VAL (if x<0)	ERANGE
	HUGE_VAL (if x>0)	ERANGE
|x|>6.74652e^9	0.0	ERANGE

creat() create a new file

TYPE UNIX I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
int creat (const char *name, int pmode);
```

DESCRIPTION

creat() creates a file and opens it for unbuffered, write-only access. If the file already exists, it is truncated to 0 length (this is done by erasing and then creating the file).

creat() returns as its value an integer called a file descriptor. Whenever a call is made to one of the unbuffered I/O functions to access the file, its file descriptor must be included in the function's parameters.

If file I/O support is desired, **creat()** must set a device table entry for the relevant file descriptor to indicate that the file descriptor is associated with an opened file. It must also enter the new file into the file system.

If file I/O support is not intended, and file descriptors will always have a fixed interpretation (i.e. '1' will refer to port 1), creat() can always return -1 and indicate an appropriate error code in errno.

See **open(C)** for more information on: file descriptors, file systems, and device table. **creat()** is not necessary to support Standard I/O.

name is a pointer to a character string which is the name of the device or file to be opened.

For most systems, *pmode* is optional; if specified, it is ignored. It should be included, however, for programs for which UNIX-compatibility is required, since the UNIX **creat()** function requires it. In this case, *pmode* should have the octal value 0666.

DIAGNOSTICS

If **creat()** fails, it returns -1 as its value and sets a code in the global integer **errno**. **errno** is set to EMFILE if all file descriptors in the device table are associated with open files.

NOTES

creat() is system dependent, and must be specially written for your system.

ctime() convert a time value to an ASCII string

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <time.h>
char *ctime (const time_t *timer);
```

DESCRIPTION

ctime() is equivalent to the **asctime()** function, except that the time value should be in **time_t** format rather than **tm** format. As with **asctime()**, a pointer to the ASCII time string is returned.

ctime() converts the time value that is pointed to by *timer* to local time. **ctime()** returns as its value a pointer to the resulting time; this is in the form of a character string. The string is contained in a static buffer that is in the **asctime()** function. **ctime(*timer*)** is equivalent to **asctime (localtime(*timer*))**

NOTES

The time value that is passed to **ctime()** is usually obtained from the system independent function **mktime()** or from the system dependent function **time()**. Thus, before you can use **ctime()**, you may have to implement **time()**.

SEE ALSO

asctime(), time(), localtime(), strftime(), gmtime()

ctop() convert string from C to Pascal format

TYPE String
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

char *ctop (char*_str_)

DESCRIPTION

Character strings are represented differently in C and Pascal: In C a string consists of the characters followed by a null character. In Pascal, a string consists of a byte containing the number of characters in the string, followed by the string.

ctop() converts a string from C to Pascal format. The converted string overlays the original string and **ctop()** returns a pointer to the converted string.

SEE ALSO

ptoc()

difftime() compute the difference between two times

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <time.h>
double difftime (time_t *time1*, time_t *time 0*);

DESCRIPTION

difftime() returns the difference of the two time values: *time1* - *time0*. The difference is expressed in seconds and is returned as a **double**.

NOTES

The time value that is passed to **difftime()** is usually obtained from the system independent function **mktime()** or from the system dependent function **time()**. Thus, before you can use **difftime()**, you may have to implement **time()**.

SEE ALSO

ctime(), clock(), mktime(), time()

div() compute quotient and remainder

TYPE Integer Math
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
div_t div (int numerator, int denominator)
```

DESCRIPTION

div() divides two signed integers and returns both the quotient and remainder as a type **div_t**. The **div_t** type is defined in the **stdlib.h** header file as being

```
typedef struct {
    int quot;
    int rem;
} div_t;
```

SEE ALSO

ldiv()

_exit() terminate calling program

TYPE Miscellaneous
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
void _exit (int code);
```

DESCRIPTION

_exit() terminates the calling program.

Unlike **exit()**, **_exit()** does not first close files open for standard I/O or call functions registered with **atexit()**.

NOTES

_exit() is system dependent, and must be specially written for your system.

SEE ALSO

exit(), **abort()**, **_abort()**, **atexit()**

exit() terminate calling program

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY **c.lib**

DECLARATION

```
#include <stdlib.h>
void exit (int code);
```

DESCRIPTION

_exit() terminates the calling program. Before terminating the program, **exit()** calls all functions registered with **atexit()**.

exit() also closes and flushes all files opened for standard I/O and deletes all files created by **tmpfile()**.

NOTES

exit() calls **_exit()**, which is system dependent. Thus, before you can use **exit()**, **_exit()** must be specially written for your system.

SEE ALSO

_exit(), abort(), _abort(), atexit()

exp() compute the exponential function e^x

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double exp (double x);
```

DESCRIPTION

exp() computes the exponential function of the input parameter *x* and returns it.

DIAGNOSTICS

Values of *x* which cause **exp()** to compute an extremely large value, i.e., greater than LOG_HUGE, will cause **exp()** to return the value HUGE_VAL and set the error code ERANGE in the globally defined integer **errno**.

Error codes:

condition	return value	errno
x>LOGHUGE	HUGE_VAL	ERANGE
x<LOGTINY	0.0	ERANGE

SEE ALSO

log(), log10(), frexp(), ldexp(), modf()

fabs() return the absolute value of a given number

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double fabs (double x);
```

DESCRIPTION

fabs() returns the absolute (or positive) value of the parameter *x*. Therefore, **fabs()** will return 5.3 for an input of either 5.3 or -5.3.

SEE ALSO

floor(), ceil(), fmod()

fclose() close a buffered I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fclose (FILE *stream);
```

DESCRIPTION

fclose() causes the specified stream to be flushed and the device or file associated with the stream to be closed. Any data that was written to the stream's output buffer but not yet written to the file or device will be written by **fclose()** before closing the file, and the input and output buffers used by the stream will be deallocated. **fclose()** is automatically called by **exit()** before exiting the program so that no devices or files are left open after the program terminates.

DIAGNOSTICS

fclose() returns 0 if it is successful. If *stream* is not a valid, open stream, EOF is returned.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fopen()

fdopen() open a file or device previously opened

TYPE UNIX I/O
COMPATIBILITY Aztec/UNIX
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
FILE *fdopen (int fd, char *mode);
```

DESCRIPTION

fdopen() is used to associate a standard I/O stream with a file or device previously opened for unbuffered I/O (with **open()** or another unbuffered I/O function.) The *mode* parameter should match the mode with which the file or device was originally opened.

The various modes and their meanings are:

Mode	Meaning
a	Open text stream for appending. If the file exists, it is positioned one character past the last character in the file. If the file does not exist, it is created. In both cases, the file is opened as write-only.
ab	Same as **a**, except the stream is opened as binary.
a+	Same as **a**, except the stream may also be read from.
a+b or ab+	Same as **a+**, except the stream is opened as binary.
r	Open text stream for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.
rb	Same as **r**, except the stream is opened as binary.
r+	Same as **r**, but the stream may also be written to.
r+b or rb+	Same as **r+**, except the stream is opened as binary.
w	Open a text stream for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created with a length of 0.
wb	Same as **w**, except the stream is opened as binary.
w+	Same as **w**, except the stream may also be read from.

w+b or **wb+** Same as **w+**, except the stream is opened as binary.

EXAMPLES

```
#include <stdio.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main(argc, argv)
char **argv;
int argc;
{
    FILE *fp;
    int fd;
    fd = open (argv[1], O_WRONLY+O_CREAT+O_EXCL);
    if (fd == -1)
    {
          if (errno == EEXIST)
                printf ("file already exists\n");
          else if (errno == ENOENT)
                printf ("unable to open file \n");
          else
                printf ("open error \n");
    } else
          printf ("the file %s was opened\n", argv[1]);
    if ((fp = fdopen(fd, "r+")) == NULL)
          printf ("can't open file for r+ \n");
    else
          printf ("fdopen worked\n");
    close (fd);
    fclose (fp);
}
```

```
#include <fcntl.h>
#include <errno.h>
main(argc, argv)
char **argv;
int argc;
{
    FILE *fp;
    int fd;
    fd = open (argv[1],
    O_WRONLY+O_CREAT+O_EXCL);
    if (fd == -1)
    {
          if (errno == EEXIST)
                printf ("file already exists\n");
          else if (errno == ENOENT)
                printf ("unable to open file \n");
          else
                printf ("open error \n");
    } else
          printf ("the file %s was opened\n",
           argv[1]);
    if ((fp = fdopen(fd, "r+")) == NULL)
          printf ("can't open file for r+ \n");
    else
          printf ("fdopen worked\n");
    close (fd);
    fclose (fp);
}
```

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fopen(), freopen()

feof() test for end-of-file in a standard I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int feof (FILE *stream);
```

DESCRIPTION

feof() is used to test for an end-of-file condition within a specified stream. **feof()** will return a non-zero value if a stream has reached an end of file; otherwise, it will return a zero. This function is necessary because the standard I/O functions return EOF not only for end-of-file conditions, but also if a read error occurs.

SEE ALSO

ferror(), clearerr(), perror()

ferror() test for an error in a standard I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int ferror (FILE *stream);
```

DESCRIPTION

ferror() is used to determine if an I/O error has occurred in the specified stream. **ferror()** will return a non-zero value if an error has occurred in the stream; otherwise, it will return a 0. This function should be used in conjunction with the **feof()** function to distinguish between end-of-file and true error conditions. An error condition will persist until **clearerr()** is called or the stream is closed.

SEE ALSO

feof(), clearerr(), perror()

fflush() flush an I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fflush (FILE *stream);
```

DESCRIPTION

fflush() is used to explicitly flush an I/O stream of any data in its buffers which has not yet been written to the file or device associated with the stream. **fflush()** is automatically called by **fclose()** and also any write operation which outputs an end-of-line sequence or causes the stream's buffer to overflow.

DIAGNOSTICS

If **fflush()** is successful, it returns 0. If a write error occurs, it returns EOF. If stream is NULL, all streams are flushed.

NOTES

Before this system independent function can be used, the system dependent, unbuffered I/O functions (**_open()**, **_close()**, **_read()**, **_write()**, and **lseek()**) must be specially written for your system.

SEE ALSO

fclose(), **fopen()**, **freopen()**, **ungetc()**

fgetc() return the next available character

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fgetc (FILE *stream);
```

DESCRIPTION

fgetc() returns the next character available from *stream* and advances the stream's file position by 1. The character is returned as an unsigned char promoted to an int. This function is identical to **getc()**, except that it is implemented as a true function rather than a macro.

DIAGNOSTICS

If an error occurs during the read operation, or if the end-of-file is reached, **fgetc()** will return EOF. The functions **feof()** and **ferror()** may be used to distinguish between a true error and end-of-file.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fopen(), fclose(), agetc(), getc(), getchar()

fgetpos() save the current file position for a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fgetpos (FILE *stream, fpos_t *pos);
```

DESCRIPTION

fgetpos() saves the current file position indicator for the specified stream into the object pointed to by *pos*. The value stored in *pos* is suitable for use by the **fgetpos()** function to reposition the stream.

fgetpos() returns 0 if successful. If it is not successful, it returns a non-zero value or sets an error code in the global integer **errno**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

ftell(), fseek(), fsetpos()

fgets() get a string of characters from a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
char *fgets (char *s, int n, FILE *stream);
```

DESCRIPTION

fgets() reads in a string of characters from the specified stream. **fgets()** will place characters from *stream* into the array pointed to by *s* until *n*-1 characters are read, a newline-sequence is reached, or the end-of-file is encountered. If a newline is reached, it is included in the string. **fgets()** will terminate the string with a null. The pointer *s* is returned by **fgets()** if no errors occur.

DIAGNOSTICS

If end-of-file is encountered before any characters are read, the array pointed to by *s* is left unchanged, and a NULL pointer is returned. If a read error occurs, the array's contents should not be considered valid, and a NULL pointer is also returned. The **ferror()** and **feof()** functions may be used to distinguish between an error and end-of-file.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

ferror(), fgetc(), getc(), getchar(), gets()

_filbuf() get and return next character

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int _filbuf(FILE *stream);
```

DESCRIPTION

The **_filbuf()** function obtains and returns the next character from the specified stream. If the end-of-file is reached, it returns EOF. Any open streams with pending output are flushed. This routine is primarily for internal library use by **stdio** routines.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

fileno() return file descriptor associated with a stream

TYPE UNIX I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fileno (FILE *stream);
```

DESCRIPTION

fileno() returns the low-level file descriptor associated with the specified stream. The file descriptor can then be used with the unbuffered I/O functions (**open(), read(), etc.**)

SEE ALSO

feof(), ferror(), clearerr(), perror()

_fileopen() open file and associate specified stream with it

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
FILE *_fileopen(const char *name, const char *mode, FILE *stream, int fd);
```

DESCRIPTION

_fileopen() opens the named file and associates the specified stream with it. If *name* is a NULL pointer, then the stream is associated with the file descriptor *fd*. The mode argument is used as the mode argument to **fopen()**. The **_fileopen()** function returns a pointer to the object controlling the stream. If the open operation fails, **_fileopen()** returns a NULL pointer. This routine is primarily for internal library use by **stdio** routines.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fopen()

floor() return largest value not greater than input

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double floor (double x);
```

DESCRIPTION

floor() returns the largest integral value not greater than the input parameter *x*. This return value is expressed as a double. For example, **floor()** would return 5.0 if passed 5.3, and -6.0 if passed -5.3.

SEE ALSO

fabs(), ceil(), fmod()

_flsbuf() flush specified open stream

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int _flsbuf(FILE *stream, int data);
```

DESCRIPTION

The **_flsbuf()** function flushes specified stream if it has been opened for writing or updating and has unwritten buffers. If data is not a -1, then data is placed in the newly flushed buffer and the buffer marked as unwritten. Otherwise, the stream is placed in the neutral state awaiting either reads or writes. The **_flsbuf()** function returns the value of the data passed in. If an error occurs writing the stream, the error flag is set in the stream and EOF is returned. If the data value is -1, a zero is returned if no errors occur. This routine is primarily for internal library use in **stdio** routines.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

fmod() return the remainder of the double value x/y

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double fmod (double x, double y);
```

DESCRIPTION

fmod() calculates the double value *x* modulo *y*. The exact remainder, called *f*, is calculated so that $x = iy + f$ for an integer *i*, and $0<f<y$.

SEE ALSO

ceil(), floor(), modf()

fopen() open a file or device for standard I/O access

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
FILE *fopen (const char *filename, const char *mode);
```

DESCRIPTION

fopen() is used to prepare, or "open", the device or file pointed to by filename for access by the standard I/O functions. Once opened by **fopen()**, a file or device is referred to as a STREAM.

If the device or file is successfully opened, **fopen()** returns a pointer, called a file pointer, to a structure of type FILE. This file pointer is then taken as a parameter by functions such as **getc()** or **putc()** to read from, write to, and generally access the stream.

The first parameter to **fopen()** is a pointer to the name of the device or file you want to open.

The other parameter passed to **fopen()**, mode, specifies how the file or device is to be accessed. The mode parameter defines two important variables concerning stream accessibility: read/write access and text/binary access.

- Read/write accessibility determines whether the file may be read from, written to, or both. Also, the initial position within the file upon opening, and course of action if the file does not exist, may be defined.

- Text/binary access determines whether the stream should be interpreted as a series of "lines" (text mode), or as raw data (binary mode). In text mode, there is no guarantee of a correspondence between the number of characters written or read and the actual position within the file, due to possible end-of-line sequence translation and other possible alterations. In binary mode, however, there is a 1:1 correspondence between characters read and the position in a file.

As their names suggest, text mode is appropriate for handling straight text input in a portable manner, whereas binary mode deals with an unaltered data stream.

mode points to a character string terminated by a null that specifies the stream's accessibility. The various modes and their meanings are:

Mode	Meaning
a	Open text stream for appending. If the file exists, it is positioned one character past the last character in the file. If the file does not exist, it is created. In both cases, the file is opened as write-only.
ab	Same as **a**, except the stream is opened as binary.
a+	Same as **a**, except the stream may also be read from.
a+b or ab+	Same as **a+**, except the stream is opened as binary.
r	Open text stream for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.
rb	Same as **r**, except the stream is opened as binary.
r+	Same as **r**, but the stream may also be written to.
r+b or rb+	Same as **r+**, except the stream is opened as binary.
w	Open a text stream for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created with a length of 0.
wb	Same as **w**, except the stream is opened as binary.
w+	Same as **w**, except the stream may also be read from.
w+b or wb+	Same as **w+**, except the stream is opened as binary.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fgetpos(), fseek(), fsetpos(), ftell(), rewind(), freopen()

format() write formatted data

TYPE Conversion
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int format (int (*func) (), const char *fmt, ...);
```

DESCRIPTION

format() is used to write formatted ASCII data by calling the function *func* repeatedly with characters. *func* should take as an argument a single character, which is a character to be output.

The *fmt* string should have the same format as the function **printf()**. In fact, **printf()** could be implemented as

```
return (format (putchar, fmt, &args);
```

See the **printf()** function for a full description of **format()**'s conversion process.

SEE ALSO

va_start(), printf()

_format() write formatted ASCII data to specified stream

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int _format(FILE *stream, const char *format, va_list varg);
```

DESCRIPTION

_format() writes formatted ASCII data to the specified stream according to the format string given. format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %.

See the description of **printf()** for details on the conversion specifications. The **_format()** function returns the number of characters transmitted, or a negative value if an output error occurred. This routine is primarily for internal library use by **stdio** routines.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

format(), printf(), fprintf(), sprintf(), va_start()

fprintf() write formatted data to an I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib, m.lib

DECLARATION

```
#include <stdio.h>
int fprintf (FILE *stream, const char *fmt, ...);
```

DESCRIPTION

fprintf() is used to write formatted ASCII data to the specified stream. The *fmt* string specifies the exact output to *stream* and also determines the number of additional arguments that are required. See the **printf()** function for complete details on the *fmt* string.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

printf(), format(), sprintf()

fputc() write a character to an I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fputc (int c, FILE *stream);
```

DESCRIPTION

fputc() takes the character *c* and writes it to the specified I/O stream. Unless an I/O error occurs, **fputc()** returns *c*. If an error does occur, **fputc()** returns EOF. **fputc()** is identical to the **putc()** function.

DIAGNOSTICS

fputc() returns EOF if an error occurs. The actual error code is placed in **errno**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

putchar(), putc()

fputs() write a string to an I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fputs (const char *str, FILE *stream);
```

DESCRIPTION

fputs() writes the null-terminated character string pointed to by *str* to the specified stream. The terminating null in *str* is not written. Unlike **puts()**, **fputs()** does not write a "\n" at the end of the string. If **fputs()** is successful, it returns a positive value. If an error does occur, **fputs()** returns EOF.

DIAGNOSTICS

EOF is returned by **fputs()** if an error occurs, with the error code set in **errno**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

puts(), fputc(), putc()

fread() read from a specified standard I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
size_t fread (void *buffer,size_t size,size_t count, FILE *stream);
```

DESCRIPTION

fread() is used to read one or more characters from *stream*. **fread()** will place into the buffer pointed to by *buffer* up to *count* items, with each item of *size* size. The position indicator for *stream* is advanced by the number of characters that were successfully read. **fread()** returns as its value the number of items (*count* if no errors occurred) successfully read from *stream*, not the number of characters.

See the **fwrite()** function for an example that uses **fread()**.

DIAGNOSTICS

fread() returns 0 or a number less than count upon end-of-file or error. The functions **feof()** and **ferror()** can be used to distinguish between the two.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fwrite()

free() deallocate a memory block

TYPE Memory Allocation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
void free (void *ptr);
```

DESCRIPTION

free() deallocates a block of memory which was previously reserved with the **malloc()**, **lmalloc()**, **calloc()**, or **realloc()** functions. This allows the space pointed at by *ptr* to be used in later attempts to allocate memory. If *ptr* is a NULL pointer, **free()** does nothing. If *ptr* does not point to a block previously allocated by **malloc()**, **calloc()**, or **realloc()**, or has already been de-allocated by a call to **free()**, the results are unpredictable.

SEE ALSO

malloc(), calloc(), realloc()

freopen() reopen a stream with a new device

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <stdio.h>
FILE *freopen (const char *_filename_,const char*_mode_, FILE *_stream_);

DESCRIPTION

freopen() is used to substitute the name or device originally associated with *stream* with the new file or device filename. **freopen()** closes the original file or device and returns *stream* as its value. **freopen()** is most often used to associate new devices or files to the pre-opened streams **stdin**, **stdout**, and **stderr**. In all other respects **freopen()** is the same as **fopen()**.

EXAMPLE

```
#include <stdio.h>
main()
{
    FILE *fp;
    fp = freopen ("dskfile", "w+", stdout);
    printf ("This message is going to dskfile\n");
}
```

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fopen()

frexp() decompose a floating point number

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double frexp (double value, int *exp);
```

DESCRIPTION

frexp() breaks a floating point number into its component mantissa and exponent. Given value, **frexp()** places the exponent-portion of value into the buffer pointed to by *exp*, and returns the mantissa component.

SEE ALSO

ldexp(), modf(), exp()

fscanf() perform formatted input conversion

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib, m.lib

DECLARATION

```
#include <stdio.h>
int fscanf (FILE *stream, const char *fmt, ...);
```

DESCRIPTION

fscanf() is equivalent to the **scanf()** function, with the exception that input is read from the specified stream rather than from **stdin**. See **scanf()** for details on format string input conversion

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

scanf(), sscanf(), strtod(), strtol(), strtoul()

fseek() reposition current location within a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fseek (FILE *stream, long int offset, int origin);
```

DESCRIPTION

fseek() sets the current position within a file opened for standard I/O. *stream* is the stream which is to be repositioned. The exact operation of **fseek()** differs depending on whether the file was opened in binary or text mode.

If the file was opened in binary mode, the new position, measured in characters from the beginning of the file, is obtained by adding the requested offset to the position specified by origin. origin may have the following values:

SEEK_SET	offset from the beginning of the file.
SEEK_CUR	offset from the current position in the file.
SEEK_END	offset from the end of the file.

Offset may be positive or negative.

If the file was opened for text mode, the use of **fseek()** is restricted. Either *offset* must equal 0, or *offset* must be a value previously returned by **ftell()**, with origin set to SEEK_SET.

fseek() will clear the end-of-file indicator for the stream and undo the effects of **ungetc()** calls if successful.

DIAGNOSTICS

fseek() returns 0 if it is successful. If an error occurs, it returns a nonzero value and sets the appropriate code in **errno**.

EXAMPLE

The following routine is equivalent to opening a file in **a+** mode:

```
#include <stdio.h>
main()
{
    FILE *fopen(), *fp;
    if ((fp = fopen("file1", "r+")) == NULL)
          fp = fopen ("file1", "w+");
    fseek (fp, 0L, 2);
          /* position 1 byte past last character */
    fwrite ("did the seek",13,1,fp);
    fclose (fp);
}
```

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

lseek(), ftell()

fsetpos() set the correct file position for a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int fsetpos (FILE *stream, const fpos_t *pos);
```

DESCRIPTION

fsetpos() sets the file position indicator for the specified stream according to the value contained in the object pointed to by *pos*. *pos* should be the value obtained by a previous **fgetpos()** call.

A successful call to **fsetpos()** will clear the end-of-file indicator for the stream as well as the effects of the **ungetc()** function on the stream. On success, **fsetpos()** returns zero. If an error occurs, it returns a non-zero value and sets an error code in the global integer **errno**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

ftell(), fseek(), fgetpos()

ftell() return the current file position within a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
long int ftell (FILE *stream);
```

DESCRIPTION

ftell() returns the current position within the specified stream. For binary streams, this represents the number of characters from the beginning of the file. For text streams, the number returned by **ftell()** may only be meaningfully used by the **fseek()** function. **ftell()** calls on a text stream do not necessarily reflect a measure of characters read or written to the stream.

DIAGNOSTICS

ftell() returns -1L if an error occurs and places the error code in **errno**. If **ftell()** is successful, it returns the current file position.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fseek(), lseek(), fgetpos()

ftoa() convert floating point number to an ASCII string

TYPE Conversion
COMPATIBILITY Aztec
LIBRARY m.lib

DECLARATION

#include <stdlib.h>
void ftoa (double *val*, char **buf*, int *precision*, int *type*);

DESCRIPTION

ftoa() converts a double-precision floating point number into an ASCII string. The parameter *val* is the number to be converted, and *buf* is the buffer where the string is to be placed. It is your responsibility to ensure that the area pointed to by *buf* is sufficiently large to handle the ASCII representation of *val*.

The precision and type parameters control the format used to convert the number. precision is used to specify the number of digits to be shown to the right of the decimal point. type specifies the printf-like format used: 0 for "E" format, 1 for "F" format, and 2 for "G" format. See the description of the **printf()** function for the details of these format specifiers.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    double val;
    val = 5.75;
    ftoa (val, buf, 2, 0);
    printf ("buf = %s\n", buf);
}
```

SEE ALSO

atof(), atoi(), atol()

fwrite() write to a specified standard I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
size_t fwrite (const void *buf, size_t size, size_t count, FILE *stream);
```

DESCRIPTION

fwrite() is used to write one or more characters to *stream*. **fwrite()** takes up to count items, each of size size, from the buffer pointed to by buf. The file position indicator for stream is advanced by the number of characters that were successfully written.

DIAGNOSTICS

If a write error occurs, **fwrite()** will return a number less than *count* and set an error code in **errno**.

EXAMPLE

This is the code for reading ten integers from *file 1* (see **fread()** for more information) and writing them again to *file 2*. It includes a simple check that there are enough two-byte items in the first file:

```
#include <stdio.h>
main()
{
    FILE, *fp1, *fp2;
    char buf[50];
    int size, count, i;
    size = 2;
    count = 10;
    for (i = 0; i < 50; i++)
          buf[i] = '\0';
    if ((fp1 = fopen("file1","r")) == NULL)
    {
          printf ("You asked me to open file1");
          printf ("but I can't\n");
    }
    if ((fp2 = fopen("file2","w")) == NULL)
    {
          printf ("You asked me to open file2");
          printf ("but I can't\n");
    }
    if (fread(buf, size, count, fp1) != count)
          printf ("Not enough integers in file1\n");
    fwrite(buf, size, count, fp2);
    fclose (fp1);
    fclose (fp2);
}
```

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fread()

_getbuf() associate a buffer with a stream

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
void _getbuf(FILE *stream);
```

DESCRIPTION

The **_getbuf()** function is an internal function that associates a buffer with a stream. **stderr** is always unbuffered and interactive files are always line buffered. The **_getbuf()** function returns no value. If it is not possible to allocate a buffer of the required size, the stream is marked as unbufferd.

getc() return next available character from a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int getc (FILE *stream);
```

DESCRIPTION

getc() returns the next character available from *stream* and advances the stream's file position by 1. The character is returned as an unsigned **char** promoted to an **int**. The **getc()** function is identical to **fgetc()**, except **getc()** is defined as a macro.

DIAGNOSTICS

If an error occurs during the read operation, or if the end-of-file is reached, **getc()** returns EOF. The functions **feof()** and **ferror()** may be used to distinguish between a true error and end-of-file.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fopen(), fclose(), agetc(), fgetc(), getchar()

getchar() return the next available character from stdin

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int getchar (void);
```

DESCRIPTION

getchar() returns the next character from the standard input stream, **stdin**. It is equivalent to the call **getc(stdin)**.

DIAGNOSTICS

If an error occurs during the read operation, or if the end-of-file is reached, **getchar()** returns EOF. The functions **feof()** and **ferror()** may be used to distinguish between the two cases.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

getc(), fgetc(), fopen(), fclose(), agetc(), putc()

getenv() get value of environment variable

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
char *getenv (const char *name);
```

DESCRIPTION

getenv() returns a pointer to the character string associated with the environment variable *name*, or a NULL pointer if the variable is not in the environment. If the name cannot be found, a NULL pointer is returned.

DIAGNOSTICS

If the specified name cannot be found within the environment, a NULL pointer is returned.

NOTES

getenv() is system dependent, and must be specially written for your system.

SEE ALSO

system()

_getiob() return next available stdio stream

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
FILE *_getiob(void);
```

DESCRIPTION

_getiob() returns the next available **stdio** stream. If there are none available, a NULL pointer is returned. This routine is primarily for internal library use by **stdio** routines.

gets() get a string of characters from stdin

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
char *gets (char *buf);
```

DESCRIPTION

gets() reads a string of characters from the standard input stream, **stdin**, into the array pointed to by *buf*. **gets()** reads in characters from **stdin** until either a newline sequence or the end-of-file is encountered. The newline sequence, if encountered, is discarded, and a NULL is written immediately after the last character. This differs from the operation of the **fgets()** function, which preserves the new line. **gets()** returns *buf* if no errors occur.

DIAGNOSTICS

If end-of-file is encountered before any characters are read, the array pointed to by *buf* is left unchanged, and a NULL pointer is returned. If an error occurs, the array's contents should be treated as invalid, and a NULL pointer is returned. The **ferror()** and **feof()** functions may be used to distinguish between an error and the end-of-file.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

ferror(), fgets(), feof()

getw() read a word from input stream

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int getw(FILE *stream);
```

DESCRIPTION

getw() uses **getc()** to read a word from the input stream pointed to by *stream*. It returns an **int** that is the word received. If an error or end-of-file occurs, EOF is returned. Since EOF is a valid return value, the **ferror()** and the **feof()** functions must be used to determine if an error really occurred when EOF is returned.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

getc()

gmtime() convert date and time

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <time.h>
struct tm *gmtime (const time_t *timer);
```

DESCRIPTION

gmtime() converts a time value that is pointed at by timer to Greenwich Mean Time. **gmtime()** breaks down the resultant time into a static **struct tm** structure and return as its value a pointer to this structure.

To relate local time to Greenwich Mean Time, **gmtime()** calls **getenv()** to get the value of the TZ environment variable. The value of TZ must be a three letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three letter name for daylight time zone. For example, the setting for New Jersey is EST5EDT.

DIAGNOSTICS

gmtime() returns a null pointer if **getenv()** can not find the TZ environment variable.

NOTES

gmtime() calls the system dependent function **getenv()**. Thus, before you can call **gmtime()**, you must implement **getenv()** at least to the point where **getenv()** can return a value for the TZ environment variable.

The time that is input to **gmtime()** is usually obtained either from the system independent function **mktime()** or from the system dependent function **time()**. Thus, before you can use **gmtime()** you may have to implement **time()**

SEE ALSO

localtime(), ctime(), asctime(), time()

index() find the first occurrence of a character in a string

TYPE String Handling
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *index (char *str,int c);
```

DESCRIPTION

The **index()** function returns a pointer to the first occurrence of *c* within the string pointed to by *str*. If the character is not found, then NULL is returned.

index() is not supported by ANSI and generally is not a portable function. For this reason the equivalent ANSI function **strchr()** should be used whenever possible.

SEE ALSO

strchr(), strrchr(), rindex()

ioctl() determine and set console mode

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
int ioctl(int fd, int code, struct sgttyb *arg);
```

DESCRIPTION

ioctl() sets and determines the mode of the console. **_ioctl()** performs the same as **ioctl()** and is provided for internal library use in case you define a version of **ioctl()** that overrides the library version. For more details see the "Console I/O" section of the **Library Overview** chapter.

NOTES

This system dependent function must be specially written for your system.

SEE ALSO

isatty()

is...() character classification functions

FUNCTIONS isalpha(), isupper(), islower(), isdigit(), isalnum(), isspace(), ispunct(), isprint(), iscntrl(), isgraph(), isxdigit()

TYPE Character Type
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <ctype.h>
int isalpha (int c);
```

DESCRIPTION

These functions test a character value to check if it is of a certain type. If **isascii()** is true for *c*, then non-zero (true) will be returned under the following conditions:

isalpha() *c* is a letter

isupper() *c* is an uppercase letter

islower() *c* is a lowercase letter

isdigit() *c* is a digit

isalnum() *c* is an alphanumeric character

isspace() *c* is a space, tab, carriage return, newline, form feed or vertical tab

ispunct() *c* is a punctuation character

isprint() *c* is a printing character, valued **0x20** (space) through **0x7e** (tilde)

iscntrl() *c* is a delete character (**0xff**) or ordinary control character (value less than **0x20**)

isascii() *c* is an ASCII character, code less than **0x100**

isgraph() tests for any printing character except a space

isxdigit() tests for any hexadecimal digit character **0-9, a-f, A-F**

Otherwise, a zero (false) is returned. These functions are implemented as both true functions and as macros. By default, functions are used. If you want to use the macro versions to improve performance you should either

- **#define** __C_MACROS__ before including **ctype.h**

or

- compile with the **-sm** or **-so** options.

SEE ALSO

toupper(), tolower()

isatty() determine if device is interactive

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
int isatty(int fd);
int _isatty(int fd)
```

DESCRIPTION

isatty() returns nonzero if the file descriptor *fd* is associated with the console, and zero otherwise. **_isatty()** performs the same as **isatty()** and is provided for internal library use in case you define a version of **isatty()** that overrides the library version. For more details see the "Console I/O" section of the **Library Overview** chapter.

SEE ALSO

ioctl()

labs() return the absolute value of a signed long

TYPE Integer Math
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
long int labs (long int x);
```

DESCRIPTION

labs() returns the absolute value of the signed long *x*.

SEE ALSO

abs()

ldexp() multiply a float by an integral power of 2

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double ldexp(double x, int exp)
```

DESCRIPTION

The **ldexp()** function returns the value of *x* times 2 raised to the power *exp*.

ldiv() compute quotient and remainder of two longs

TYPE Integer Math
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <stdlib.h>
ldiv_t ldiv (long int *numerator*, long int *denominator*);

DESCRIPTION

The **ldiv()** function divides two signed long integers and retains both the quotient and remainder as a type **ldiv_t**. The **ldiv_t** type is defined in the **stdlib.h** header file as being

```
typedef struct {
    long int quot;
    long int rem;
};
```

SEE ALSO

div()

localeconv() set components of object for formatting

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <locale.h>
struct lconv *localeconv(void);
```

DESCRIPTION

The **localeconv()** function sets the components of an object with type *struct lconv* with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale. The members of the structure with type **char *** are pointers to strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type char are nonnegative numbers, any of which can be CHAR_MAX to indicate that the value is not available in the current locale.

The **localeconv()** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **localeconv()** function. In addition, calls to the **setlocale()** function with categories LC_ALL, LC_MONETARY, or LC_NUMERIC may overwrite the contents of the structure.

SEE ALSO

setlocale()

localtime() convert a time value relative to local time

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <time.h>
struct tm *localtime (const time_t *timer);
```

DESCRIPTION

localtime() converts the timepointed at by *timer* to local time.

localtime() breaks down the resulting time into a static *struct tm* structure, and returns as its value a pointer to this structure.

NOTES

The time value that is input to **localtime()** is usually obtained either fromthe system independent function **mktime()** or from the system dependent function **time()**. Thus, before you can use **localtime()** you may have to implement **time()**.

SEE ALSO

gmtime(), ctime(), asctime(), time()

log() compute the natural logarithm of a number

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double log(double x);
```

DESCRIPTION

log() returns as its value the natural logarithm of the input number *x*.

DIAGNOSTICS

If the input parameter *x* is negative, **log()** returns -HUGE_VAL and sets **errno** to EDOM. If the input value is 0, -HUGE_VAL is returned and **errno** is set to ERANGE.

Error codes:

condition	return value	errno
x = = 0	-HUGE_VAL	ERANGE
x < 0	-HUGE_VAL	EDOM

SEE ALSO

exp(), log10(), pow(), sqrt()

log10() compute the logarithm of a number to base 10

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY **m.lib**

DECLARATION

```
#include <math.h>
double log10 (double x);
```

DESCRIPTION

log10() returns the logarithm to base 10 of the input parameter *x*.

DIAGNOSTICS

If the input parameter *x* is negative, **log10()** will return -HUGE_VAL and set **errno** equal to ENOM. If the input value x is 0, -HUGE_VAL is returned, and **errno** is set to ERANGE.

Error codes:

condition	return value	errno
x = = 0	-HUGE_VAL	ERANGE
x < 0	-HUGE_VAL	EDOM

SEE ALSO

exp(), log(), pow(), sqrt()

longjmp() execute a non-local goto

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <setjmp.h>
void longjmp (jmp_buf env, int retval);
```

DESCRIPTION

A call to the **longjmp()** function restores the stack and register state saved in the last call to **setjmp()** with the argument *env*, and then executes a return which makes it appear that **setjmp()** returned the value **retval**.

It is crucial that **setjmp()** be called with env before any **longjmp()** calls occur. If **setjmp()** is not called with *env* before the first **longjmp()**, then the results are completely unpredictable and could cause serious problems.

After completion of the **longjmp()** call, program execution will continue as if the corresponding **setjmp()** call had returned *retval*. If *retval* is set to 0, then it will be forced to zero so as not to conflict with the 0 returned by the initial call to **setjmp()**.

SEE ALSO

setjmp()

lseek() change current position within file

TYPE UNIX I/O
COMPATIBILITY Aztec /UNIX
LIBRARY c.lib

DECLARATION

long lseek (int *fd*, long *offset*, int *origin*);
long _lseek (int *fd*, long *offset*, int *origin*);

DESCRIPTION

lseek() sets the current position of a file that opened for unbuffered I/O. This position determines where the next character will be read or written.

fd is the file descriptor associated with the file. The current position is set to the location specified by the *offset* and *origin* parameters, as follows:

- If *origin* is 0, the current position is set to *offset* bytes from the beginning of the file.
- If *origin* is 1, the current position is set to the current position plus *offset*.
- If *origin* is 2, the current position is set to the end of the file plus *offset*.

The offset can be positive or negative, to position after or before the specified origin, respectively. If **lseek()** is used on a file opened as text, only an offset of 0 may be used.

If **lseek()** is successful, it returns the new position in the file (in bytes from the beginning of the file).

_lseek() performs the same as **lseek()** and is provided for internal library use in case you define a version of **lseek()** that overrides the library version.

DIAGNOSTICS

If **lseek()** fails, it returns -1 as its value and sets an error code in the global integer **errno**. **errno** is set to EBADF if the file descriptor is invalid. It will be set to EINVAL if the *offset* parameter is invalid or if the requested position is before the beginning of the file.

EXAMPLES

1. To seek to the beginning of a file:

```
lseek(fd, 0L, 0);
```

lseek() returns the value zero (0) since the current position in the file is character (or byte) number zero.

2. To seek to the character following the last character in the file:

```
pos = lseek(fd, 0L, 2);
```

The variable **pos** contains the current position of the end-of-file, plus one.

3. To seek backward five bytes:

```
lseek(fd, -5L, 1);
```

The third parameter, `1`, sets the origin at the current position in the file. The offset is -5. The new position is the *origin* plus the *offset*. So the effect of this call is to move backward a total of five characters.

4. To skip five characters when reading in a file:

```
read(fd, buf, count);
lseek(fd, 5L, 1);
read (fd, buf, count);
```

NOTES

lseek() and **_lseek()** are system dependent, and must be specially written for your system.

malloc() allocate a block of system memory

TYPE Memory Allocation
COMPATIBILITY ANSI
LIBRARY **c.lib**

DECLARATION

```
#include <stdlib.h>
void *malloc (size_t size);
```

DESCRIPTION

malloc() allocates a block of memory *size* bytes long. The block is allocated from an area in system memory called the HEAP. See the **Library Overview** chapter for a description of how the heap is used.

The memory allocation done by **malloc()** is called DYNAMIC ALLOCATION, because the amount of memory to be reserved for storage is determined dynamically at runtime, rather than being fixed at compile time. The storage returned from **malloc()** should not be assumed to have been initialized to any particular value, 0 or otherwise.

If the allocation is successful, **malloc()** returns a pointer to the requested block.

DIAGNOSTICS

If **malloc()** cannot allocate the requested size block, it returns a NULL pointer. Otherwise, a pointer to the requested block is returned.

SEE ALSO

calloc(), realloc(), free()

mblen() determine size of multibyte character

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

DESCRIPTION

If *s* is not a NULL pointer, **mblen()** determines the number of bytes comprising the multibyte character pointed to by *s*. **mblen()** is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

except that it does not affect the shift state.

If *s* is a NULL pointer, **mblen()** returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a NULL pointer, the **mblen()** function either returns zero (if s points to the null character), returns the number of bytes that comprise the multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

SEE ALSO

mbtowc(), wctomb(), mbstowcs(), wcstombs()

mbstowcs() convert sequence of multibyte characters

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

DESCRIPTION

mbstowcs() converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding codes, and stores not more than *n* codes into the array pointed to by *pwcs*. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to **mbtowc()**, except that the shift state of the **mbtowc()** function is not affected.

No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

If an invalid multibyte character is encountered, **mbstowcs()** returns **(size_t)-1**. Otherwise, **mbstowcs()** returns the number of array elements modified, not including a terminating zero code, if any.

SEE ALSO

mblen(), mbtowc(), wctomb(), wcstombs()

mbtowc() determine size of multibyte character

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

DESCRIPTION

If *s* is not a NULL pointer, **mbtowc()** determines the number of bytes that comprise the multibyte character pointed to by *s*. It then determines the code for value of type wchar_t that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and *pwc* is not a NULL pointer, the **mbtowc()** function stores the code in the object pointed to by *pwc*. At most *n* bytes of the array pointed to by *s* will be examined.

If *s* is a NULL pointer, **mbtowc()** returns a nonzero or a zero value. If the multibyte character encoding has state-dependent encodings, a zero value is returned; otherwise nonzero is returned. If *s* is not a NULL pointer, the **mbtowc()** function either returns zero (if *s* points to the null character), or returns the number of bytes that comprises the converted multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns minus one (if they do not form a valid multibyte character).

In no case will the value returned be greater than *n* or the value of the MB_CUR_MAX macro.

SEE ALSO

mblen(), mbstowcs(), wctomb(), wcstombs()

memccpy() copy characters from source to destination

TYPE Block Operations
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <string.h>
void *memccpy (void *dest, const void *src, int c, size_t n);
```

DESCRIPTION

memccpy() copies characters from *src* to *dest*, stopping after the first occurrence of character *c* has been copied or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *dest*, or a NULL pointer if *c* was not found in the first *n* characters of *src*. Overlapping moves are unpredictable.

SEE ALSO

memcpy()

memchr() find a character within an object

TYPE Block Operation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
void *memchr (const void *obj, int c, size_t n);
```

DESCRIPTION

memchr() searches the first *n* bytes in the object pointed to by obj for the character *c*. If *c* is found, **memchr()** returns a pointer to it. Otherwise, a NULL pointer is returned.

SEE ALSO

strrchr(), strchr()

memcmp() compare two blocks of memory n bytes long

TYPE Block Operation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <string.h>
int memcmp (const void *_blk1_, const void *_blk2_, size_t _n_);

DESCRIPTION

The **memcmp()** function compares the first *n* characters in the object pointed to by *blk1* with the first *n* characters in the object pointed to by *blk2*. **memcmp()** returns a positive number, negative number, or zero, depending on whether *blk1* is greater than, less than, or equal to *blk2*.

SEE ALSO

strcmp(), strncmp(), strcoll()

memcpy() copy block of bytes from one object to another

TYPE Block Operation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
void *memcpy (void *dest, const void *src, size_t n);
```

DESCRIPTION

memcpy() copies the first *n* bytes from the object pointed to by *src* into the object pointed to by *dest*. The two objects should not overlap. **memcpy()** returns *dest*.

SEE ALSO

memmove(), strcopy(), strncpy(),memccpy

memmove() copy a block of memory

TYPE Block Operation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
void *memmove (void *dest, const void *source, size_t n);
```

DESCRIPTION

memmove() copies a block of data from one location in memory to another location. **memmove()** will copy *n* characters from the object pointed to by *source* to the object pointed to by *dest*. **memmove()** acts as if the data pointed to by *source* is first copied into a temporary buffer before moving it into *dest*. Therefore, overlapping blocks may be used with this function. **memmove()** returns the value of *dest*.

SEE ALSO

memcpy()

memset() set a block of memory to a specified value

TYPE Block Operation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
void memset (void *obj, int c, size_t n);
```

DESCRIPTION

memset() copies the value of *c* into the first *n* bytes of the object pointed to by *obj*.

SEE ALSO

memcpy(), memcmp()

mktime() convert a time value between formats

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#includes <time.h>
time_t mktime (struct tm *_timeptr_);

DESCRIPTION

mktime() translates a time value represented in **struct tm** format into **time_t** format. *timeptr* should be a pointer to the **tm format** time value, and the corresponding **time_t** value is returned by **mktime()**. If *timeptr* is unconvertible, **mktime()** returns **(time_t) -1**.

SEE ALSO

asctime(), ctime(), localtime(), time()

modf() break a floating point value into parts

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double modf (double val, double *iptr);
```

DESCRIPTION

modf() breaks the floating point number *val* into its integral (to the left of the decimal point) and fractional (to the right of the decimal point) components. The integral portion is stored in the buffer pointed to by *iptr*, and the fractional portion is returned as the return value of **modf()**.

SEE ALSO

frexp(), ldexp()

movmem() copy a memory block

TYPE Block Operation
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

void movmem(const void ****source*, **void ****dest*, **size_t *length*)

DESCRIPTION

movmem() copies length characters from the block of memory pointed to by *source* to the block of memory pointed to by *dest*. The effect of **movmem()** is as if *length* bytes of source were copied to a distinct temporary area and then the temporary area copied to *dest*. **movmem()** can thus be used to copy blocks of memory that overlap.

APPLICATION NOTE

movmem() is provided for compatibility with other Manx C compiler products. It should not be used in new programs. The **memmove()** function should be used instead.

SEE ALSO

memcpy(), memmove()

open() open device or file for unbuffered I/O

TYPE UNIX I/O
COMPATIBILITY Aztec /UNIX
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
open (char *name, int mode);
_open (char *name, int mode);
```

DESCRIPTION

open() opens a device or file for unbuffered I/O. It returns an integer value called a file descriptor which is used to identify the file or device in subsequent calls to unbuffered I/O functions.

name is a pointer to a character string which is the name of the device or file to be opened.

mode specifies how your program intends to access the file. The choices are as follows:

Mode	Meaning
O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read and write
O_CREAT	create file, then open it
O_TRUNC	truncate file, then open it
O_EXCL	cause open() to fail if file already exists; used with O_CREAT
O_APPEND	position file for appending data
O_BINARY	don't translate characters during I/O
O_TEXT	perform system dependent end of line translations during I/O (e.g. '\n' to '\r\n' on output '\r\n' to '\n' on input)

These **open()** modes are integer constants defined in the file **fcntl.h**. Although the true values of these constants can be used in a given call to **open()**, use of the symbolic names ensures compatibility with UNIX and other systems.

The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, or O_RDWR in the *mode* parameter. The three remaining values are optional. They may be included by adding them to the *mode* parameter, as in the examples below.

By default, the open fails if the file to be opened does not exist. To cause the file to be created when it does not already exist, specify the O_CREAT option. If O_EXCL is given in addition to O_CREAT, the open will fail if the file already exists; otherwise, the file is created.

If the O_TRUNC option is specified, the file will be truncated so that nothing is in it. The truncation is performed by simply erasing the file, if it exists, and then creating it. So it is not an error to use this option when the file does not exist.

Note that when O_TRUNC is used, O_CREAT is not needed.If O_APPEND is specified, the current position for the file (that is, the position at which the next data transfer will begin) is set to EOF. This option is not supported by UNIX.

If file I/O support is desired, **open()** must set a device table entry for the relevant file descriptor to indicate that the file descriptor is associated with an opened file. If the O_CREAT or O_TRUNC option is specified, it must also enter the new file into the file system. File position information must also be recorded in the device table.

If file I/O support is not intended, and file descriptors will always have a fixed interpretation (i.e. '1' will refer to port 1), **open()** can be a dummy function.

If **open()** does not detect an error, it returns an integer called a file descriptor. This value is used to identify the open file during unbuffered I/O operations. The file descriptor is very different from the file pointer which is returned by **fopen()** for use with buffered I/O functions.

_open() performs the same as **open()** and is provided for internal library use in case the user defines a version of **open()** that overrides the library version.

DIAGNOSTICS

If **open()** encounters an error, it returns -1 and places a code in the global integer **errno**. **errno** is set to ENOENT if the file does not exist and O_CREAT was not specified. It is set to EEXIST if the file exists and (O_CREAT+O_EXCL) was specified. If an invalid file descriptor is passed, **errno** is set to EMFILE.

EXAMPLES

1. To open the file **testfile** for read-only access:

```
fd = open("testfile", O_RDONLY);
```

If **testfile** does not exist **open** returns -1 and sets errno to ENOENT.

2. To open the file **testfile** for read-write access:

```
fd = open("sub1", O_RDWR+O_CREAT);
```

If the file does not exist, it will be created and then opened.

3. The following program opens a file whose name is given on the command line. The file must not already exist.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main(argc, argv)
char **argv;
int argc;              /* should add this declaration */
{
    int fd;
    fd = open (argv[1],O_WRONLY+O_CREAT+O_EXCL);
    if (fd == -1){
          if (errno == EEXIST)
                printf ("file already exists\n");
          else if (errno == ENOENT)
                printf ("unable to open file \n");
          else
                printf ("open error \n");
    }
    else
          printf ("the file %s was opened\n", argv[1]);
    close (fd);
}
```

NOTES

_open() and **open()** are system dependent, and must be written specially for your system.

SEE ALSO

clearerr(), ferror(), feof(), strerror()

perror() print a system error message

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
void perror (const char *str);
```

DESCRIPTION

When an error occurs in one of the standard math or I/O library functions, an error number indicating which error occurred is written to the global integer **errno**. The **perror()** function is used to print an error message to the **stderr** stream which corresponds to the error indicated by **errno**. The exact format of the message written to **stderr** is:

- The string pointed at by *str* (only if *str* and the first character it points to are not null) followed by a colon and a space.
- The system error message string followed by a new-line.

The possible values of **errno** and their corresponding error messages are:

Errno Value	Message
0	Error 0
ENOENT	No such file or directory
E2BIG	Arg list too long
EBADF	Bad file descriptor
ENOMEM	Not enough memory
EEXIST	File exists
EINVAL	Invalid argument
ENFILE	File table overflow
EMFILE	Too many open files
ENOTTY	Not a console
EACCESS	Permission denied
EIO	I/O error
ENOSPC	No space left on device
ERANGE	Result too large
EDOM	Argument out of domain
ENOEXEC	exec() format error
EROFS	Read-only file system
EXDEV	Cross-device rename
EAGAIN	Nothing to read

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

peek...(),poke...() get and set bytes in memory

FUNCTIONS peekb(), peekl(), peekw(), pokeb(), pokel(), pokew()
TYPE Memory Allocation
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
void pokeb (unsigned long addr, int val);
int peekb (unsigned long addr);
void pokew (unsigned long addr, int val);
int peekw (unsigned long addr);
void pokel (unsigned long addr, long val);
long peekl (unsigned long addr);
```

DESCRIPTION

These functions get and set one, two, or four bytes located anywhere in memory.

addr is a pointer to the field to be modified.

peekb() returns as its value the byte at the target location.

peekl() returns as its value the contents of the four-byte field pointed at by *addr*.

peekw() returns as its value the word (that is, two bytes) at the target location.

pokeb() sets the byte at the target location to *val*.

pokel() sets the 4-byte field pointed at by *addr* to *val*.

pokew() sets the word at the target location to *val*.

pow() compute x to the yth power

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double pow (double x, double y);
```

DESCRIPTION

Given two double precision numbers *x* and *y*, **pow()** returns the value of *x* to the *y*th power (x^y).

DIAGNOSTICS

If **x = 0** and **y ≤ 0**, ENOM is set for **errno** and 0.0 is returned. If **x < 0** and **y** is not an integral value, EDOM is set for **errno** and 0.0 is returned.

SEE ALSO

exp(), log(), log10(), sqrt()

printf() formatted output function

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib, m.lib

DECLARATION

```
#include <stdio.h>
int printf (const char *fmt,...);
```

DESCRIPTION

printf() is the C language's primary formatted-output function. **printf()** takes a series of arguments, converts them to ASCII strings, and writes the formatted information to the **stdout** stream.

The Format String

The format string *fmt* must be present in every call to **printf()**. This string determines exactly what gets written to **stdout** and may contain two types of items, ordinary characters and conversion specifiers:

- Ordinary characters are always copied verbatim to the output stream.
- Conversion specifiers direct **printf()** to take arguments from **printf()**'s argument list and format them. Conversion specifiers always begin with a % character.

Conversion Specifiers

Conversion specifiers always take the form

% [flags] [width] [.precision] [size-mod] type

where

- The optional *flags* field controls output justification, sign characters on numerical values, prefixes on hex and octal numbers, decimal points, and trailing blanks.
- The optional *width* field specifies the minimum number of characters to print (the field width), with padding done with blanks or zeros.
- The optional *precision* field specifies either the minimum number of digits to be printed for integral types, or the number of characters after the decimal point to be printed for floating-point types.
- The optional *size-mod* field specifies that the argument may be long, short, or unsigned.

- The *type* field specifies the actual type of the argument that **printf()** will be converting, such as string, integer, double, etc.

Note that all of the above fields are optional except for *type*. The following tables list the valid options for these fields. In the table, integral types are considered to be **char, int, long int, short int,** and unsigned versions of these types. Floating-point types are **float, double,** and **long double**.

Flags Field

Character	Effect on Conversion
-	The result is left justified, with padding on the right with blanks. By default, when "-" is not specified, the result is right-justified with padding on the left with 0's or blanks.
+	The result will always have a "-" or "+" prepended to it if it is a numeric conversion.
space	Positive numbers begin with a space instead of a "+" character, but negative values still have a prepended "-".
#	The argument is formatted using an alternate form from the usual conversion. For **x** or **X** types, a 0x or 0X is used as a prefix to the argument. For **0** types, a 0 is always prepended to non-zero results. For **e**, E, and **F** types, the result will always contain a decimal point, even when the number has no numbers following the decimal point. **G** and **g** types are also converted this way, with the addition that trailing zeros are not removed.

Width Field

Character	Effect on Conversion
n	A minimum of *n* characters are output. If the conversion has less than *n* characters, the field is padded with blanks.
*	The *width* specifier is supplied in the argument list before the actual conversion argument.

Precision Field

Character	Effect on Conversion
.*n*	A minimum of *n* characters will be provided in the field for integral conversions (**d, i, o, u, x, X**). For floating-point conversions (**e, E, f, g, G**) *n* specifies the number of digits after the decimal point. For string conversions (**s**), *n* is the number of characters printed from the string.

.	This is the same as .*n* where *n* is 0. No decimal point is printed.
.*	The precision is supplied in the argument list before the actual conversion argument.

Size-mod Field

Character	Effect on Conversion
h	The argument should be taken as a short integer. Valid only for integral conversion (**d, i, o, u, x, X**).
l	The argument should be taken as a long **int** for integral conversions (**d, i, o, u, x, X**) and as a **double** for floating-point conversions (**e, E, f, g, G**).
L	The argument should be taken as a long **double** for floating-point conversions (**e, E, f, g, G**).

type Field

Character	Argument	TypeConversion
d	integral type	signed decimal integer (base 10)
i	integral type	signed decimal integer
o	integral type	unsigned octal (base 8) integer
u	integral type	unsigned decimal integer (base 16)
x	integral type	unsigned hexadecimal integer with lower case letters, i.e. **a, b, c, d, e, f**
X	integral type	unsigned hexadecimal integer with uppercase letters, i.e., **A, B, C, D, E, F**
f	floating point	signed real number with the form **[-]*ddd.ddd***. The number of digits after the decimal point is determined by the *precision* field, or is 6 if precision is not specified. The "." will not appear if precision is 0.
e	floating point	Signed real number with the form **[-]*d.ddd e+dd***. There is always exactly one digit before the decimal point, followed by an **e**, followed by an exponent at least two characters long. Precision considerations are the same as **f**.
E	floating point	Same as **e**, but with a capital **E** for the exponent.

g	floating point	Signed real number in either **f** or **e** form. **e**-format is used if the exponent is less than -4 or greater than or equal to the precision (6 by default). Otherwise, **f** format is used.
G	floating point	Same as **g**, but use **E** or **f** formats.
c	integral type	The integer argument is converted to an unsigned **char**, and the ASCII character corresponding to the number is output.
s	pointer	The string pointed to is written out until either a NULL is reached, or the number of characters written equals the *precision* field.
p	pointer	The contents of the pointer is written to the output.
n	pointer	The number of characters written so far is written to the argument, which should be a pointer to an **int**. No output conversion is done.
%		A % is written, and no argument is converted. The full syntax of this is %%.

There are two versions of **printf()** in the libraries: a non-floating point version in **c.lib** and a floating point version in **m.lib**. If a **%f** or **%g** conversion prints out as **f** or **g**, then you are either not linking in a math library or are linking libraries in the wrong order. To fix this problem, you should link in the math library before the C library on your link line, as shown:

```
ln68 ... -lm -lc
```

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fprintf(), sprintf(), vfprintf(), vsprintf()

ptoc() convert string from Pascal to C format

TYPE String
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

char* ptoc (char*)

DESCRIPTION

ptoc() converts a string from Pascal to C format. The converted string overlays the original string, and **ptoc()** returns a pointer to the converted string.

SEE ALSO

ctop()

putc() write a character to an I/O stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int putc (int c, FILE *stream);
```

DESCRIPTION

putc() takes the character *c* and writes it to the specified I/O stream. Unless an I/O error occurs, **putc()** returns *c*. If an error does occur, **putc()** returns EOF. This function is identical to **fputc()**.

DIAGNOSTICS

putc() returns EOF if an error occurs. The actual error code is placed in **errno**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

putchar(), fputc(), puts(), putw()

putchar() write a character to the stdout stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int putchar (int c);
```

DESCRIPTION

putchar() is identical to the function **putc()**, except that it always writes to **stdout**, i.e., it is the same as

```
putc (stdout)
```

DIAGNOSTICS

putchar() returns EOF if an I/O occurred during the write. The error code is set in the global integer **errno**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

putc(), fputc(), puts(), putw()

puts() write a string to stdout

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int puts (const char *str);
```

DESCRIPTION

puts() writes the null-terminated character string pointed to by *str* to the **stdout** stream, followed by a \n. The null at the end of the string is not written to **stdout**. If **puts()** is successful, it returns a positive value. If an error occurs, EOF is returned.

DIAGNOSTICS

EOF is returned by **puts()** if an error occurs, with the error number contained in **errno**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

putc(), putchar(), putw(), fputs()

putw() calls putc() to output word to the stream

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int putw(int w, FILE *stream);
```

DESCRIPTION

putw() calls **putc()** to output the word *w* to the stream pointed to by *stream*. The **putw()** function returns the argument *w* as its value or EOF if an error occurs while writing to the output stream. Since EOF is a valid argument to the **putw()** function, the caller must use the **ferror()** or the **feof()** function to determine if an error has really occurred when EOF is returned.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

putc(), putchar(), puts()

qsort() sort an array of records in memory

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
void qsort (void *array, size_t num,size_t width, int (*compar)
(const void *, const void *));
```

DESCRIPTION

The **qsort()** function sorts an array of *nmemb* objects, the initial element of which is pointed to by *base*. The size of each object is specified by size.

The contents of the array are sorted into ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified

EXAMPLE

The Aztec linker, ln68, can generate a file of text containing a symbol table for a program. Each line of the file contains an address at which a symbol is located, followed by a space, followed by

the symbol name. The following program reads such a symbol table from the standard input, sorts it by address, and writes it to standard output.

```
#include <stdio.>
define MAXLINES 2000
define LINESIZE 16
char *lines[MAXLINES];
main()
{
          int i, numlines, cmp();
          char buf[LINESIZE],
          * malloc(), *gets();
          for (numlines = 0; numlines < MAXLINES; ++numlines)
          {
                if (gets(buf) == (char *)NULL)
                      break;
                lines[numlines] = malloc(LINESIZE);
                strcpy(lines[numlines], buf);
          }
          qsort (lines, numlines,sizeof(char *), cmp);
          for (i = 0; i < numlines; ++i)
                printf ("%s \n", lines[i]);
}
cmp(a, b)
char **a, **b;
{
          return strcmp(*a, *b);
}
```

raise() send signal to executing program

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <signal.h>
int raise(int sig);
```

DESCRIPTION

raise() sends the signal *sig* to the executing program. It returns zero if successful, and nonzero if not.

NOTES

raise() calls the system dependent function **_exit()**. Thus, before you can use **raise()** you must write **_exit()**.

SEE ALSO

signal();

ran() generate floating point random numbers

TYPE Float Math
COMPATIBILITY Aztec
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double ran (void);
```

DESCRIPTION

ran() returns as its value a random floating point number between 0.0 and 1.0. The *seed* value for **ran()** can be set with the function **sran()**.

ran() is not defined by ANSI.

SEE ALSO

sran(), rand(), srand()

rand() return a pseudo-random integer

TYPE Integer Math
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int rand (void);
```

DESCRIPTION

rand() returns a pseudo-random integer in the range 0 to RAND_MAX. The seed value used by **rand()** may be set with the **srand()** function.

SEE ALSO

srand(), ran(), sran()

randl() return a random number

TYPE Float Math
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <locale.h>
double randl(double x);
```

DESCRIPTION

randl() returns a random number between 0.0 and 1.0. It is an alternate to **ran()** in that it uses the *x* arg in generating the next random number.

SEE ALSO

ran(), sran()

read() read from a device or file using unbuffered I/O

TYPE UNIX I/O
COMPATIBILITY Aztec /UNIX
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
int read (int fd, void *buf, size_t bufsize);
int _read (int fd, void *buf, size_t bufsize);
```

DESCRIPTION

read() reads characters from a device or disk file which has been previously opened by a call to **open()** or **creat()**. In most cases, the information is read directly into the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer into which the information is to be placed.

bufsize is the number of characters to be transferred.

If **read()** is successful, it returns as its value the number of characters transferred.

If the returned value is zero, then end-of-file has been reached, immediately, with no bytes read.

If the device or file was explicitly opened with a call to **open()**, *fd* is the file descriptor which was returned by **open()**. **creat()** cannot be used to open a file for reading.

If the returned value is greater than zero but less than *bufsize*, end-of-file has been reached after reading the returned number of bytes.

_read() performs the same as **read()** and is provided for internal library use in case you define a version of **read()** that overrides the library version.

DIAGNOSTICS

If the operation is not successful, **read()** returns -1 and places a code in the global integer **errno**. **errno** is set to EBADF if the file descriptor is invalid. It is set to EINVAL if the file or device associated with the file descriptor is only permits write access.

NOTES

read() and **_read()** are system dependent and must be specially written for your system.

SEE ALSO

open(C), close(C),write(C), Unbuffered I/O (O), Errors(O)

realloc() re-allocate memory block to a different size

TYPE Memory Allocation
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
void *realloc (void *ptr, size_t size);
```

DESCRIPTION

realloc() changes the size of a memory block, *ptr*, that was originally allocated with the **calloc()**, **realloc()**, or **malloc()** functions. The data contained in the original block is guaranteed to be preserved by **realloc()**, even if the block has to be moved to accommodate a larger size.

If *size* is larger than the original block size, the area beyond the original block size should not be assumed to contain any initial value. If the new size is smaller, the data will be truncated at the appropriate point. If *size* is 0, **realloc()** deallocates the block in a manner similar to the **free()** function. If *ptr* is 0, **realloc()** behaves like the **malloc()** function and allocates a new block of length *size*.

If *ptr* does not point to a block previously allocated by **malloc()**, **calloc()**, or **realloc()**, or if *ptr* was deallocated by the **free()** function, the behavior of **realloc()** is unpredictable.

DIAGNOSTICS

If **realloc()** cannot find a block at least *size* bytes in length available, it returns a NULL pointer. Otherwise, a pointer to the requested block is returned.

SEE ALSO

malloc(), calloc(), free()

remove() delete a file

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int remove (const char *filename);
```

DESCRIPTION

remove() deletes the disk file named *filename* from the disk. **remove()** returns 0 if it is successful, and non-zero if it is not.

DIAGNOSTICS

If an error occurs, **remove()** sets **errno** with the error code and returns a non-zero value.

SEE ALSO

unlink()

rename() rename a disk file

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int rename (const char *old, const char *new);
#include <fcntl.h>
int _rename (const char *old, const char *new);
```

DESCRIPTION

rename() changes the name of a file. *old* is a pointer to a null-terminated string containing the old file name, and *new* is a pointer to a null-terminated string containing the new name of the file.

If successful, **rename()** returns 0 as its value; if unsuccessful, it returns EOF.

If a file with the new name already exists, **rename()** sets E_EXIST in the global integer **errno** and returns EOF as its value without renaming the file.

_rename() performs the same as **rename()** and is provided for internal library use in case you define a **rename()** that overrides the library version.

NOTES

rename() and **_rename()** are system dependent, and must be specifically written for your system.

rewind() reposition a stream's position indicator

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
void rewind (FILE *stream);
```

DESCRIPTION

rewind() sets the indicated stream's file position indicator to the beginning of the file. **rewind()** is equivalent to the call

```
(void) fseek (stream, 0L, SEEK_SET);
```

with the exception that **rewind()** also clears the error indicator for the stream.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fseek()

rindex() find last occurrence of a character in a string

TYPE String Handling
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *rindex (char *str, int c);
```

DESCRIPTION

rindex() returns a pointer to the last occurrence of *c* within the string pointed to by *str*. If the character is not found in *str*, then NULL is returned.

rindex() is not supported by ANSI and generally is not a portable function. For this reason, the equivalent ANSI function **strrchr()** should be used whenever possible.

SEE ALSO

strchr(), strrchr(), index()

sbrk() Increment a pointer by *size* bytes

TYPE Memory Allocation
COMPATIBILITY Aztec/UNIX
LIBRARY c.lib

DECLARATION

```
#include <fcntl.h>
void *sbrk(size_t size)
```

DESCRIPTION

sbrk() provides an elementary means of allocating space from the heap. More sophisticated buffer management schemes can be built using this function; for example, the standard functions **malloc()**, **free()**, etc. call **sbrk()** to get heap space, which they then manage for the calling functions. Because the buffered I/O functions in stdio are in turn built using **malloc()** and **free()**, **sbrk()** must be implemented in order for these routines to be used in an application.

sbrk() increments or decrements a pointer, called the 'heap pointer', by *size* bytes, and, if successful, returns the value that the pointer had on entry. The heap pointer initially points at the base of the heap.

SEE ALSO

The functions **malloc()**, **free()**, etc, implement a dynamic buffer-allocation scheme using the **sbrk()** function. See the "Dynamic Buffer Allocation" section of the **Library Overview** chapter for more information.

The standard I/O functions usually call **malloc()** and **free()** to allocate and release buffers for use by I/O streams. This is discussed in the "Standard I/O" section of the **Library Overview** chapter.

sbrk() calls **brk()** to set the heap's HIGH WATER mark; that is the pointer to the top of allocated heap space.

Your program can safely mix calls to the malloc functions, standard I/O calls, and calls to **sbrk()** and **brk()**, as long as the your calls to **sbrk()** and **brk()** don't decrement the heap pointer. Mixing **sbrk()** and **brk()** calls that decrement the heap pointer with calls to the **malloc()** functions and/or the standard I/O functions is dangerous and probably shouldn't be done by normal programs.

ERRORS

sbrk() returns -1 if an error occurs, and set the global integer **errno** to ENOMEM.

_scan() convert text characters from input stream

TYPE Standard I/O
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int _scan(FILE *stream, const char *format, va_list varg);
```

DESCRIPTION

_scan() converts text characters from the specified input stream as directed by the format string and places the results in pointer parameters that follow. The format is composed of zero or more directives; one or more white-space characters; an ordinary multibyte character (not %); or a conversion specification. Each conversion specification is introduced by the character %.

For a detailed description of the format string see **scanf()**. **_scan()** returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the **_scan()** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure. This routine is primarily for internal library use by **stdio** routines.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

scanf(), fscanf(), sscanf(), va_start()

scanf() formatted input conversion on stdin stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY **c.lib, m.lib**

DECLARATION

```
#include <stdio.h>
int scanf (const char *fmt, ...);
```

DESCRIPTION

scanf() converts a stream of text characters from the stream as directed by the control string pointed to by the *fmt* parameter and places the results in the additional pointer parameters. There must be the same number of format specifiers in the *fmt* string as there are pointer arguments.

The Format String

The *fmt* string controls exactly how **scanf()** will scan, convert, and store each of the input fields. The conversion string is made up of the following items:

- conversion specifiers
- white space (spaces, tabs, newlines)
- ordinary characters

The **scanf()** function works through the *fmt* string, attempting to match each control item with a portion of the input stream. During the matching process, **scanf()** fetches characters one at a time from the input.

When a character is fetched which is not appropriate for the item being matched, **scanf()** pushes the character back into the stream using **ungetc()**.

scanf() terminates when it first fails to match an item in the *fmt* string or when the end of the input stream is reached. It returns the number of matched conversion specifiers or EOF if the end of the input stream was reached.

Matching White Space Characters

When a white space character is encountered in the control string, the **scanf()** function fetches input characters until the first non-white-space character is read. The non-white-space character is pushed back into the input and the **scanf()** function proceeds to the next item in the control string.

Matching Ordinary Characters

If an ordinary character is encountered in the control string, the **scanf()** function fetches the next input character. If it matches the ordinary character, the **scanf()** function simply proceeds to the next control string item. If it does not match, the **scanf()** function terminates.

Matching Conversion Specifications

When a conversion specification is encountered in the control string, the **scanf()** function first skips leading white space on the input stream or buffer. It then fetches characters from the stream or buffer until encountering one that is inappropriate for the conversion specification. This character is pushed back into the input.

If the conversion specification did not request assignment suppression (discussed below), the character string which was read is converted to the format specified by the conversion specification, the result is placed in the location pointed at by the current pointer argument, and the next pointer argument becomes current. The **scanf()** function then proceeds to the next control string item.

If assignment suppression was requested by the conversion specification, the **scanf()** function simply ignores the fetched input characters and proceeds to the next control item.

Details of Input Conversion

A conversion specification consists of:

- The character %, which tells the **scanf()** function that it has encountered a conversion specification
- Optionally, the assignment suppression character *
- Optionally a field width, that is, a number specifying the maximum number of characters to be fetched for the conversion
- A conversion character, specifying the type of conversion to be performed.

If the assignment suppression character is present in a conversion specification, the **scanf()** function will fetch characters as if it were going to perform the conversion, ignore them, and proceed to the next control string item.

The following conversion characters are supported:

%	A single % is expected in the input. No assignment is done.
d	A decimal integer is expected, the input digit string is converted to binary and the result placed in the **int** field pointed at by the current pointer argument. The corresponding argument is pointer to **int**.

o An octal integer is expected; the corresponding pointer should point to an **int** field in which the converted result will be placed. The corresponding argument is pointer to unsigned **int**.

x A hexadecimal integer is expected; the converted value will be placed in the **int** field pointed at by the current pointer argument. The corresponding argument is pointer to unsigned **int**.

s A sequence of characters delimited by white space characters is expected; they, plus a terminating null character, are placed in the character array pointed to by the current pointer argument.

c A character is expected. It is placed in the **char** field pointed at by the current pointer to character array. The normal skip over leading white space is not done; to read a single **char** after skipping leading white space, use **%1s**. The field width parameter is ignored, so this conversion can be used only to read a single character. It matches a sequence of characters of the number specified by field width.

[A sequence of characters, optionally preceded by white space but not terminated by white space, is expected. The input characters, plus a terminating null character, are placed in the character array pointed at by the current pointer argument. The left bracket is followed by:

optionally, a ^ or ~ character
a set of characters
a right bracket,].

If the first character in the set is not ^ or ~, the set specifies characters which are allowed; characters are fetched from the input until one is read which is not in the set.

If the first character in the set is ^ or ~, the set specifies characters which are not allowed; characters are fetched from the input until one is read which is in the set.

i a signed integer, value of 0 for base argument. Corresponding argument should be **ptr** to **int**.

u unassigned decimal integer. The argument corresponding to this item should be **ptr** to unsigned **int**.

p reads in a hexadecimal long value which represents a pointer. The corresponding argument should be a void pointer.

n Argument should be a **ptr** to an **int**. The number of characters read in so far from **stdin** is written to this pointer. Execution of **%n** does not increment the assignment count returned at completion of execution of the **fscanf()/scanf()** function.

e, f, g A floating point number is expected. The input string is converted to floating point format and stored in the float field pointed at by the current pointer argument. The input format for floating point numbers consists of an optionally

signed string of digits, possibly containing a decimal point, optionally followed by an exponent field consisting of an E or **e** followed by an optionally signed digit.

The conversion characters **d**, **o**, and **x** can be capitalized or preceded by **l** to indicate that the corresponding pointer is to a long rather than an **int**. Similarly, the conversion characters **e** and **f** can be capitalized or preceded by **l** to indicate that the corresponding pointer is to a **double** rather than a **float**.

The conversion characters **o**, **x**, and **d** can be optionally preceded by **h** to indicate that the corresponding pointer is to a **short** rather than an **int**.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fscanf(), sscanf()

setbuf() associate an I/O stream with a specific buffer

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <stdio.h>
void setbuf (FILE *_stream_, char *_buf_);

DESCRIPTION

setbuf() is equivalent to the **setvbuf()** function called in the following manner:

- If *buf* is not null, then **setbuf()** is the same as **setvbuf()** called with _IOFBF (fully buffered) for *mode* and BUFSIZE (defined in **stdio.h**) for *size*. Note that this means that your buffer MUST be at least BUFSIZE in length.
- If *buf* is null, then **setbuf()** is the same as **setvbuf()** called with _IONBF (no buffering) for *mode*.

SEE ALSO

setvbuf()

setjmp() set up for a non-local goto

TYPE Save Calling Environment Macro
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

DESCRIPTION

setjmp() is used in conjunction with **longjmp()** to allow gotos between functions. This is useful for error recovery from low-level routines as well as quick returns from deeply-nested functions.

The argument *env* should be declared as an instance of the type **jmp_buf**. **setjmp()** saves the current stack and register variable information in *env* so that this information may be restored upon invocation of a **longjmp()**, and returns a 0.

SEE ALSO

longjmp()

setlocale() select appropriate portion of program's locale

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <locale.h>
char *setlocale(int *category*, const char **locale*);

DESCRIPTION

The **setlocale()** function selects the appropriate portion of the program's locale as specified by the *category* and *locale* arguments. The **setlocale()** function may be used to change or query the program's entire current locale or portions thereof.

The value LC_ALL for *category* names the program's entire locale; the other values for *category* name only a portion of the program's locale.

LC_COLLATE affects the behavior of the **strcoll()** and **strxfrm()** functions.

LC_CTYPE affects the behavior of the character handling functions and the multibyte functions.

LC_MONETARY affects the monetary formatting information returned by the **localeconv()** function.

LC_NUMERIC affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the non-monetary formatting information returned by the **localeconv()** function.

LC_TIME affects the behavior of the **strftime()** function.

A value of "C" for locale specifies the minimal environment for C translation; a value of "" for locale specifies the implementation-defined native environment.

At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

If a pointer to a string is given for *locale* and the selection can be honored, the **setlocale()** function returns the string associated with the specified category for the new locale. If the selection cannot be honored, the **setlocale()** function returns a NULL pointer and the program's locale is not changed.

A NULL pointer for *locale* causes the **setlocale()** function to return the string associated with the category for the program's current locale; the program's locale is not changed.

The string returned by the **setlocale()** functions is such that a subsequent call with that string and its associated category will restore that part of the program's locale. The string returned shall not be modified by the program, but may be overwritten by a subsequent call to the **setlocale()** function. Currently the only locale supported is "C" for C program translation.

SEE ALSO

localeconv()

setmem() copy value of char into object

TYPE Block operations
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <string.h>
void setmem(void *s, size_t n, int c);
```

DESCRIPTION

setmem() copies the value of *c* (converted to an unsigned **char**) into each of the first *n* characters of the object pointed to by *s*.

APPLICATION NOTE

setmem() is provided for compatibility with other Manx C compilers. It should not be used in new programs; use **memset()** instead.

setvbuf() associate an I/O stream with a specific buffer

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY **c.lib**

DECLARATION

#include <stdio.h>
int setvbuf (FILE *_stream_, char *_buf_, int _mode_, size_t _size_);

DESCRIPTION

setvbuf() is used when you want to explicitly assign a buffer to an I/O stream, rather than let the standard I/O system do it automatically. **setvbuf()** should be called after the stream has been opened, but before any I/O functions (i.e., reading and writing) have been performed on the stream.

The type of buffering to be performed on the stream is determined by the *mode* argument:

_IOFBF The stream is FULLY BUFFERED. On reads from the stream, the I/O system will attempt to fill the entire buffer if the buffer is empty before returning the requested byte(s). On writes, the buffer is written to the file only if the buffer is full.

_IOLBF The stream is LINE BUFFERED. As with _IOFBF, reads from the stream will fill the entire buffer. Writes, however, will flush the buffer if a newline is encountered as well as if the buffer is full.

_IONBF The stream is UNBUFFERED, and the *buf* and *size* arguments are ignored. This means that each read operation will read directly from the file, and each written operation will write directly to the file, with no intermediate buffering done.

You must insure that *buf* stays in existence throughout the time the stream is open. This means that if *buf* is a local array, the stream must be closed before the function returns. Also, you are responsible for deallocating *buf* if it was dynamically allocated; it is not done automatically by **fclose()**. If you do deallocate *buf*, it must be deallocated after **fclose()**.

SEE ALSO

setbuf()

signal() define how to handle a signal

TYPE Signal
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <signal.h>
void (*signal (int sig, void (*func) (int) ) ) (int);
```

DESCRIPTION

signal() specifies that the signal whose number is *sig* is to be handled as defined by *func*. A SIGNAL is a special asynchronous event such as an operator-initiated interrupt or an arithmetic fault.

Signals and the *sig* Parameter

The following list defines the symbolic values that *sig* can have and the signal associated with each value. These values are defined in **signal.h**. The signals that are currently supported are:

SIGFPE traps divide by 0 and overflow

SIGILL traps on illegal instruction

SIGSEGV traps illegal address or bus error

Signal Processing and the *func* Parameter

func defines the action to be performed on receipt of the specified signal. It can be one of three values: SIGN_DFL, SIG_IGN, or a function address.

If the *func* for a signal is SIG_DFL, the program will be terminated by the operating system, without execution of the normal Aztec C exit code. This could result in a loss of information in files opened for standard output.

If the *func* for a signal is SIG_IGN, the signal will be ignored.

Any other value for *func* is assumed to be the address of a function. In this case, when the specified signal occurs, the function will be called, passing the signal's number as the function's only argument. Before the function is called, the value of *func* for the received signal will be set to SIG_DFL.

The function associated with a signal can terminate its program, if desired, by calling **exit()** or **longjmp()**. It can also return to the program at the point of interruption by issuing a return statement.

Return Values from Signal

If a requested change is accepted, signal returns the value that the specified signal's *func* had on entry to signal. If the change is rejected, signal returns the value SIG_ERR and the global integer **errno** is set to indicate the error. Currently, the only cause for rejection is an invalid signal number, causing **errno** to be set to EINVAL.

If the request cannot be honored, a value of SIG_ERR is returned and a positive value is stored in **errno**.

EXAMPLES

The following program calls **signal()**, so that upon receipt of a bus error the program can shut itself down in an orderly fashion, closing opened files, deleting temporary files, and so on.

```
#include <signal.h>
main()
{
    signal(SIGSEGV, shutdown)
    ... /* normal program execution */
}
shutdown(sign)
int sig;
{
    print("received signal %d\n", sig);
    ... /* termination */
    exit(1);
}
```

SEE ALSO

raise(), abort(), exit()

sin() return the sine of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double sin (double x);
```

DESCRIPTION

sin() returns the sine of *x*. *x* should be specified in radians.

Error Codes:

condition	return value	errno
$\|x\| > 6.7465e^9$	0.0	ERANGE

SEE ALSO

acos(), asin(), cos()

sinh() return the hyperbolic sine of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double sinh (double x);
```

DESCRIPTION

sinh() returns the hyperbolic sine of *x*. **sinh()** will return HUGE_VAL and set **errno** equal to ERANGE if *x* is greater than 348.6.

Error codes:

condition	return value	errno
|*x*| > LOGHUGE+ ~0.6932	HUGE_VAL	ERANGE

SEE ALSO

cosh()

sprintf() write formatted data to a buffer

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY **c.lib, m.lib**

DECLARATION

```
#include <stdio.h>
int sprintf (char *buf, const char *fmt, ...);
```

DESCRIPTION

sprintf() is used to write formatted ASCII data to the specified buffer *buf*. The *fmt* string specifies the exact contents of *buf* and also determines the number of additional required arguments. See the **printf()** function for complete details on the *fmt* string.

SEE ALSO

printf(), fprintf(), format()

sqrt() compute non-negative square root of a value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double sqrt (double x);
```

DESCRIPTION

sqrt() returns the nonnegative square root of the input parameter *x*.

DIAGNOSTICS

If *x* is negative, **errno** is set to EDOM and **sqrt()** returns 0.0 as its value.

SEE ALSO

exp(), log(), log10(), pow()

sran() set the random number seed for ran

TYPE Float Math
COMPATIBILITY Aztec
LIBRARY m.lib

DECLARATION

```
#include <math.h>
void sran (long seed);
```

DESCRIPTION

sran() is used to set the *seed* value for the function **ran()**.

sran() is not defined by ANSI.

SEE ALSO

ran(), rand(), srand()

srand() Initialize the seed value used by rand

TYPE Integer Math
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
void srand (unsigned int seed);
```

DESCRIPTION

The value *seed* passed to **srand()** is used to determine the pseudo-random sequence returned by future calls to **rand()**. If **srand()** is called more than once with the same value for *seed*, then the same sequence will be repeated. If **rand()** is called before **srand()**, then **rand()** will behave as if **srand()** had been called with a *seed* of 1.

SEE ALSO

rand(), ran(), sran()

sscanf() perform formatted input conversion on buffer

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib, m.lib

DECLARATION

```
#include <stdio.h>
int sscanf (const char *str, const char *fmt, ...);
```

DESCRIPTION

sscanf() is identical to the **scanf()** function, except that **sscanf()** reads from the buffer pointed to by *str* rather than from **stdin**. See **scanf()** for details on *fmt* string

SEE ALSO

scanf(), fscanf()

_stkchk() performs stack depth checking

TYPE Miscellaneous
COMPATIBILITY Aztec
LIBRARY stkchk.c

DECLARATION

DESCRIPTION

The **_stkchk()** function performs stack-depth checking. It is called automatically on entry to functions that have been compiled with **c68**'s **-bd** option. Source for **_stkchk()** is in **stkchk.c**, in directory **\lib\rom68**. Before using it, you must customize it.

strcat() concatenate two strings together

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <string.h>
char *strcat(char *_dest_, const char *_src_);

DESCRIPTION

strcat() appends a copy of the string pointed to by *src* to the string pointed to by *dest*, so that the resulting length of *dest* is **strlen** (*dest*) + **strlen** (*src*). The first character in *src* will overwrite the ending null in *dest*. **strcat()** will then return a pointer to *dest*.

It is important to note that the area pointed to by *dest* must be large enough to hold both the strings in *dest* and *src* (**strlen** (*dest*) + **strlen** (*src*) +1) and it is your responsibility to ensure this.

SEE ALSO

strncat(), strcpy(), strncpy()

strchr() search for first occurrence of string character

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strchr (const char *str, int c);
```

DESCRIPTION

strchr() returns a pointer to the first occurrence of the character *c* in string *str*. If *c* is not contained in *str*, then **strchr()** returns a NULL *str*.

The **strchr()** function is equivalent to the Aztec function **index()**, with the exception that **strchr()** can match a null character, whereas **index()** cannot.

SEE ALSO

strrchr(), **index()**, and **rindex()**

strcmp() compare two strings

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
int strcmp (const char *str1, const char *str2);
```

DESCRIPTION

strcmp() compares the strings pointed to by *str1* and *str2*, and determines whether *str1* is greater than, less than, or equal to *str2*. Equality is determined in the following manner:

- **strcmp()** will perform an unsigned comparison on each character in *str1* and *str2*, starting with the first character in each and advancing by one character at a time. **strcmp()** will stop when it finds two differing characters or it reaches the end of one of the strings.
- If the end of *str1* is reached but not the end of *str2*, then *str1* is less than *str2*.
- If the end of *str2* is reached before *str1*, then *str1* is greater than *str2*.
- If the end of both strings is reached simultaneously with no differing characters, then the two strings are equal.
- If the two characters differ, then *str1* is greater than *str2* if *str1*'s character is greater than *str2*. Otherwise, *str1* is less than *str2*.

It should be remembered that **strcmp()** does numerical comparisons, not character comparisons, so that the character "a" (ASCII 97) is considered greater than the character "Z" (ASCII 90).

The value returned by **strcmp()** is:

- less than 0 (negative) if *str1* is less than *str2*.
- equal to 0 if *str1* is equal to *str2*.
- greater than 0 (positive) if *str1* is greater than *str2*.

SEE ALSO

strncmp()

strcoll() compare two strings using the current locale

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
int strcoll (const char *str1, const char *str2)
```

DESCRIPTION

The **strcoll()** function is equivalent to the **strcmp()** function, with the exception that **strcoll()** will use the current locale for character translation. (**strcoll()** is truly equivalent to **strcmp()**, as Aztec C currently supports only the C locale.)

SEE ALSO

strcmp()

strcpy() copy one string to another

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <string.h>
char *strcpy (char *_dest_, const char *_src_);

DESCRIPTION

strcpy() copies the characters in the string pointed to by *src* to the area pointed to by *dest*, including the terminating null character. The area pointed to by *dest* must be at least (**strlen**(*src*) + 1) characters long. **strcpy()** always returns *dest*.

SEE ALSO

strncpy(), memcpy(), memmove(), strlen()

strcspn() return the index of specified string

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
size_t strcspn (const char *str1, const char *str2);
```

DESCRIPTION

The **strcspn()** function returns the index of the first character in the string pointed to by *str1* which matches a character in the string pointed to by *str2*. This index is equal to the number of initial characters in *str1* which do not match a character in *str2*.

SEE ALSO

strspn(), strpbrk(), strstr(), strtok()

strdup() copy the string pointed to

TYPE String Handling
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strdup(char *_s);
```

DESCRIPTION

strdup() makes a copy of the string pointed to by *s*. The space for the copy is allocated using the **malloc()** function. **strdup()** returns a pointer to the array allocated. If the array could not be allocated, a NULL pointer is returned.

SEE ALSO

malloc()

strerror() maps error number to error message

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strerror(int errnum);
```

DESCRIPTION

strerror() maps the error number in *errnum* to an error message string. **strerror()** returns a pointer to the string, the contents of which correspond to a description for the *errnum* passed in. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **strerror()** function.

strftime() place characters into array pointed to

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <string.h>
size_t strftime(char *s, size_t *maxsize*, const char **format*, const struct tm **timeptr*);

DESCRIPTION

strftime() places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The format string consists of zero or more conversion specifications and ordinary multibyte characters. A conversion specification consists of a % character followed by a character that determines the conversion specification's behavior. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. No more than *maxsize* characters are placed into the array. Each conversion specification is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the program's locale and by the values contained in the structure pointed to by *timeptr*.

%a	the locale's abbreviated weekday name.
%A	the locale's full weekday name.
%b	the locale's abbreviated month name.
%B	the locale's full month name.
%c	the locale's appropriate date and time representation.
%d	the day of the month as a decimal number (01-31).
%H	the hour (24-hour clock) as a decimal number (00-23).
%I	the hour (12-hour clock) as a decimal number (01-12).
%j	the day of the year as a decimal number (001-366).
%m	the month as a decimal number (01-12).
%M	the minute as a decimal number (00-59).
%p	the locale's equivalent of either AM or PM.
%S	the second as a decimal number (00-59).
%U	the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
%w	the weekday as a decimal number [0 (Sunday)-6].
%W	the week number of the year (Monday as the first day of the week) as a decimal number (00-53).
%x	the locale's appropriate date representation.
%X	the locale's appropriate time representation.
%y	the year without century as a decimal number (00-99).
%Y	the year with century as a decimal number.
%Z	the time zone name, or by no characters if no time zone is determinable.
%%	the % character.

If a conversion specification is not one of the above, the behavior is undefined.

strftime() returns the number of characters placed into the array pointed to by *s* not including the terminating null character if the total number of resulting characters including the terminating null character is not more than *maxsize*. Otherwise, zero is returned and the contents of the array are indeterminate.

NOTES

For the **%Z** conversion **strftime()** calls the system dependent function **getenv()** to get the value of the TZ environment variable, see the description of the **gmtime()** function. Thus, before you can have **strftime()** do a **%Z** conversion, you must implement **getenv()** at least to the point where it can return the value of the TZ environment variable.

strlen() return the length of a string

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
size_t strlen (const char *str);
```

DESCRIPTION

strlen() returns the number of characters in the string pointed to by *str*, not including the terminating null character.

You should be careful in allocating space for strings based on **strlen()**. A common mistake is to code the following:

```
ptr = malloc (strlen ("STRING");
strcpy (ptr, "STRING");
```

This will only set aside six characters for STRING, when seven are actually needed (the six characters in the word plus the ending null), and the **strcpy()** that follows the **malloc()** call will write too many characters.

The correct way to use **strlen()** is

```
ptr = malloc (strlen ("STRING") +1);
strcpy (ptr, "STRING");
```

SEE ALSO

strcpy(), strncpy()

strncat() concatenate two strings together

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strncat (char *dest, const char *src, size_t max);
```

DESCRIPTION

strncat() appends a copy of the string pointed to by *src* to the string pointed to by *dest* until either *max* characters have been appended or a null is reached in *src*, whichever comes first. The maximum resulting length of the string pointed to by *dest* will be **strlen** (*dest*) and *max*. The first character in *src* overwrites the ending null in *dest*. **strncat()** always returns a pointer to *dest*.

It is your responsibility to ensure that the area pointed to by *dest* is at least (**strlen**(*dest*) + *max* + 1) characters long.

SEE ALSO

strcat(), strcpy(), strncpy()

strncmp() compare two strings, up to *max* characters

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <string.h>
int strncmp (const char *_str1_, const char *_str2_, size_t _max_);

DESCRIPTION

strncmp() is equivalent to the **strcmp()** function, with the exception that **strncmp()** will only compare a maximum of *max* characters. **strncmp()** returns a positive number, negative number, or zero, depending on whether *str1* is greater than, less than, or equal to *str2*. See the **strcmp()** function for more details.

SEE ALSO

strcmp(), memcmp(), strcoll(), strxfrm()

strncpy() copy characters from one string to another

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strncpy (char *dest, const char *src, size_t max);
```

DESCRIPTION

strncpy() copies a maximum of *max* characters from the string pointed to by *src* to the area pointed to by *dest*. If *src* is less than *max* characters long, then *dest* is padded with null characters until *max* total characters have been written. The area pointed to by *dest* must be at least *max* characters in length.

SEE ALSO

strcpy(), memcpy(), memove(), strlen()

strpbrk() return pointer to first character in a string

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strpbrk (const char *str1, const char *str2);
```

DESCRIPTION

The **strpbrk()** function returns a pointer to the first character in the string pointed to by *str1* which is contained in the string pointed to by *str2*. Null is returned if no characters within *str1* match those in *str2*.

SEE ALSO

strspn(), strcspn(), strstr(), strtok()

strrchr() search for occurrence of character in string

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strrchr (const char *str, int c);
```

DESCRIPTION

strrchr() returns the last occurrence of the character *c* in string *str*. If *c* is not contained in *str*, then **strrchr()** returns a NULL.

The **strrchr()** function is equivalent to the Aztec function **rindex()**, with the exception that **strrchr()** can match a null character, whereas **rindex()** cannot.

SEE ALSO

strchr(), **index()**, and **rindex()**

strspn() return index of the first character in a string

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
size strspn (const char *str1, const char *str2);
```

DESCRIPTION

The **strspn()** function returns the index of the first character in *str1* which is not contained in the string pointed to by *str2*. This index is equal to the number of initial characters in *str1* which match characters in *str2*.

SEE ALSO

strcspn(), strpbrk(), strstr(), strtok()

strstr() return a pointer to first occurrence of a string

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strstr(const char *str1, const char *str2);
```

DESCRIPTION

strstr() returns the first occurrence of the substring *str2* contained within the string *str1*. If *str2* is not contained within *str1*, then NULL is returned.

SEE ALSO

stcspn(), strspn(), strpbrk(), strtok()

strtod() convert a string to a double

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <stdlib.h>
double strtod (cont char *_nptr_, char **_endptr_);

DESCRIPTION

strtod() converts the initial portion of the string pointed to by *nptr* into a double value, which is returned by **strtod()**. The string must be in the form

[*ws*] [+ | -] *ddd* [.] [*ddd*] [*exp*]

where

- *ws* is whitespace (newline, space, tab)
- + | - means + or -
- *ddd* are digits 0-9
- *exp* is an optional exponent of the form

  [e | E + | - *ddd*]

  where the vertical lines, |, indicate "or".

strtod() stops reading characters when it encounters a character in *nptr* which cannot be interpreted as part of a double value. It sets **endptr* equal to a pointer to this character (as long as *endptr* is not NULL).

DIAGNOSTICS

strtod() will return HUGE_VAL if the string causes overflow.

SEE ALSO

atof(), stroul(), strtol(), atol(), atoi()

strtok() tokenize a string

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
char *strtok (char *str, const char *del_list);
```

DESCRIPTION

strtok() is designed to be called multiple times to break *str* into a series of tokens, each of which is delimited by a character contained in *del_list*. The operation of **strtok()** is detailed below.

First Invocation

On the first invocation of **strtok()** with *str*, **strtok()** will search for the first character in *str* which is not contained in *del_list* (that is, it scans past delimiters.)

- If a character is found which is not in *del_list*, it is considered to be the start of the first token.
- If such a character is not found, there are no tokens in *str* and a NULL pointer is void.

strtok() then searches from the token's start for a character that IS contained in *del_list* (it scans to the next delimiter.)

- If a character is found in *str* which is contained in *del_list*, **strtok()** overwrites it with a null character which terminates the token. **strtok()** will then internally save a pointer to the following character, from which the next token search will start. A pointer to the token is then returned by **strtok()**.
- If no characters in *str1* are found to be in *del_list*, then the current token is considered to extend to the end of *str*. **strtok()** will return a pointer to the token, and subsequent calls to **strtok()** will return a NULL pointer.

Subsequent Invocation

All calls of **strtok()** following the initial call should pass a NULL pointer for the first argument. This instructs **strtok()** to use its internal pointer in order to locate the next token. **strtok()** will then behave as described above. Subsequent calls may, if you wish, have different delimiters specified by *del_list*.

SEE ALSO

strchr(), strrchr(), strspn(), strcspn()

strtol() convert a string to a long

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

DESCRIPTION

strtol() converts the initial portion of the string pointed to by *nptr* to long **int** representation. First it decomposes the input string into three parts:

- an initial, possibly empty, sequence of white-space characters (as specified by the **isspace()** function)
- a subject sequence resembling an integer represented in some radix determined by the value of *base*
- a final string of one or more unrecognized characters, including the terminating null character of the input string.

Then it attempts to convert the subject sequence to an integer and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant using normal C syntax to specify the base, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest subsequence of the input string, starting with the first non-white-space character, that is an initial subsequence of a sequence of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit. If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to normal C syntax rules.

If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence

begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

strtol() returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value would cause overflow, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and the value of the macro ERANGE is stored in **errno**.

strtold() convert a string to a long double

TYPE Conversion
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
long double strtold(const char *nptr, char **endptr);
```

DESCRIPTION

strtold() converts the intitial portion of the string pointed to by *nptr* to long double representation. First it decomposes the input string into three parts:

- an initial, possibly empty, sequence of white-space characters (as specified by the **isspace()** function)
- a subject sequence resembling a floating-point constant
- a final string of one or more unrecognized characters, including the terminating null character of the input string.

Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point character, then an optional exponent part, but no floating suffix. The subject sequence is defined as the longest subsequence of the input string, starting with the first non-white space character, that is an initial subsequence of a sequence of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

strtold() returns the converted value, if any. If no conversion is performed, zero is returned. If the correct value would cause overflow, plus or minus HUGE_VAL is returned (according to the sign of the value), and the value of the macro ERANGE is stored in **errno**. If the correct value would cause underflow, zero is returned and the value of the macro ERANGE is stored in **errno**.

strtoul() convert a string to unsigned long integer

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

#include <stdlib.h>
unsigned long int strtoul(const char ****nptr*, char *****endptr***, int** ***base*****);**

DESCRIPTION

strtoul() converts the initial portion of the string pointed to by *nptr* to unsigned long **int** representation. First it decomposes the input string into three parts

- an initial, possibly empty, sequence of white-space characters (as specified by the **isspace()** function)
- a subject sequence resembling an unsigned integer represented in some radix determined by the value of *base*
- a final string of one or more unrecognized characters, including the terminating null character of the input string.

Then it attempts to convert the subject sequence to an integer and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant using normal C syntax to specify the base, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest subsequence of the input string, starting with the first non-white-space character, that is an initial subsequence of a sequence of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to normal C syntax rules. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to

the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

strtoul() returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value would cause overflow, ULONG_MAX is returned, and the value of the macro ERANGE is stored in **errno**.

strxfrm() transform a string to match the current locale

TYPE String Handling
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <string.h>
size_t strxfrm (char *dest, const char *src, size_t n);
```

DESCRIPTION

strxfrm() transforms the string pointed at by *src* to match the current locale. The transformed string is then copied to *dest*, and the length of *dest* is returned.

Because only the **c** locale is currently supported, **strxfrm()** actually does no transformations, but simply performs a **strcpy()** followed by a **strlen()**.

SEE ALSO

strcpy(), strcoll(), strcmp(), strlen()

swapmem() swap characters between specified objects

TYPE Block Operations
COMPATIBILITY Aztec
LIBRARY c.lib

DECLARATION

```
#include <string.h>
void swapmem(void *s1, void *s2, size_t n);
```

DESCRIPTION

swapmem() swaps *n* characters between the object pointed to by *s1* and the object pointed to by *s2*. If swapping takes place between objects that overlap, the behavior is undefined.

system() make call to underlying operating system

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int system (const char *string);
```

DESCRIPTION

system() passes the string pointed to by *string* to the host environment to be executed by a "command processor" in an implementation defined manner. In the current implementation no command processor support is provided, and the routine simply returns 0.

NOTES

system() is system dependent and must be specially written for your system.

SEE ALSO

abort(), atexit(), exit(), getenv()

tan() return the tangent of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double tan(double x);
```

DESCRIPTION

tan() returns the tangent of *x*. *x* should be specified in radians.

Error codes:

condition	return value	errno
$\|x\| .. 6.74652e^9$	0.0	ERANGE

SEE ALSO

atan(), atan2()

tanh() return the hyperbolic tangent of a double value

TYPE Float Math
COMPATIBILITY ANSI
LIBRARY m.lib

DECLARATION

```
#include <math.h>
double tanh (double x);
```

DESCRIPTION

tanh() returns the hyperbolic tangent of *x*.

time() return the time of day

TYPE Time
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <time.h>
time_t time(time_t *timeptr);
```

DESCRIPTION

The **time()** function returns as its value the current time. If *timeptr* is not null, this value is also assigned to the location pointed at by *timeptr*.

If the current time is not available, **time()** returns -1.

NOTES

time() is system dependent, and must be specially written for your system.

SEE ALSO

ctime(), gmtime(), localtime()

tmpfile() create a temporary file

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
FILE *tmpfile(void);
```

DESCRIPTION

tmpfile() creates a temporary binary file and opens it for standard I/O in update **wb+** mode. **tmpfile()** returns as its value the file's **FILE** pointer.

When the temporary file is closed, either because the program explicitly closes it or because the program terminates, the temporary file is automatically deleted.

If the file cannot be created, the function returns a NULL pointer.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

tmpnam(), mktemp(), fopen()

tmpnam() create a name for a temporary file

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
char *tmpnam (char *s);
```

DESCRIPTION

tmpnam() creates a character string that can be used as the name of a temporary file and returns as its value a pointer to the string. The generated string is not the name of an existing file.

s optionally points to an area into which the name will be generated. This must contain at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in **stdio.h.**

s can also be a NULL pointer. In this case, the name will be generated in an internal array. The contents of this array are destroyed each time **tmpnam()** is called with a null argument.

tmpnam() can be called a maximum of TMP_MAX times.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

tmpfile(), mktemp()

tolower() convert a character to lowercase

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <ctype.h>
int tolower (int c);
```

DESCRIPTION

tolower() converts an uppercase character to lowercase. If the value of *c* is an uppercase character, **tolower()** returns its lowercase equivalent, otherwise *c* is returned unchanged.

SEE ALSO

toupper()

toupper() convert a character to uppercase

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <ctype.h>
int toupper(int c);
```

DESCRIPTION

toupper() converts a lowercase character to uppercase. If the value of the argument *c* is a lowercase character, **toupper()** returns its uppercase equivalent as its value, otherwise *c* is returned unchanged.

SEE ALSO

tolower()

ungetc() push a character back into input stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

DESCRIPTION

ungetc() pushes the character value of the argument *c* back onto an input stream. That character will be returned by the next **getc()** call on that stream. **ungetc()** returns *c* as its value.

Only one character of pushback is guaranteed. EOF cannot be pushed back.

DIAGNOSTICS

ungetc() returns EOF (-1) if the character cannot be pushed back.

NOTES

Before this system independent standard I/O function can be used, the system dependent unbuffered I/O functions must be specially written for your system.

SEE ALSO

fgetpos(), fseek(), fsetpos(), ftell()

unlink() erase file

TYPE UNIX I/O
COMPATIBILITY Aztec/UNIX
LIBRARY c.lib

DECLARATION

int unlink (const char *_name_);
int _unlink (const char *_name_);

DESCRIPTION

unlink() erases a file, where *name* is a pointer to a character array containing the name of the file to be erased.

unlink() returns a 0 value if successful.

Since **unlink()** is not defined by ANSI, it is strongly recommended that the equivalent ANSI function **remove()** be used in its place.

_unlink() performs the same as **unlink()** and is provided for internal library use in case you define a version of **unlink()** that overrides the library version.

DIAGNOSTICS

unlink() returns -1 if it could not erase the file and places a code in the global integer **errno** describing the error.

NOTES

unlink() and **_unlink()** are system dependent, and must be specially written for your system.

SEE ALSO

creat()

va_...() variable argument access

FUNCTIONS va_arg(), va_end(), va_start()
TYPE Variable Arguments
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdarg.h>
type va_arg (va_list argptr, type);
void va_end (va_list argptr);
void va_start (va_list argptr, parmN);
```

DESCRIPTION

These three macros are used as a mechanism to access individual arguments from within a function that accepts a variable number of arguments.

A function which receives a variable argument count should declare a variable, *argptr*, of *type* **va_list**. This variable will be used as a pointer to the current argument being processed.

The **va_start()** macro should be called first to initialize *argptr* to point to the first argument in the list. the *parmN* parameter should be the last fixed argument passed to the function.

Once **va_start()** has been called, **va_arg()** may be called repeatedly to fetch arguments sequentially from the argument list. The *type* parameter to **va_arg()** should be the C-data *type* of the next argument (i.e., **int**, lens, **char** *x*, etc.). **va_arg()** returns the value of the argument.

When variable-argument processing is completed, the **va_end()** macro should be called.

vfprintf() write formatted ASCII data to a stream

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf (FILE *stream, const char *fmt, va_list args);
```

DESCRIPTION

vfprintf() is equivalent to the **fprintf()** function, except that the variable argument list is replaced by *args*. *args* should be initialized by the **va_start()** macro before the call to **vfprintf()** is made.

The return value of **vfprintf()** is equal to the number of characters written or is a negative value if a write error occurs.

SEE ALSO

vprintf(), vsprintf(), printf(), va_start()

vprintf() write formatted ASCII data to stdout

TYPE Standard I/O
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdarg.h>
#include <stdio.h>
int vprintf (const char *fmt, va_list args)
```

DESCRIPTION

vprintf() is equivalent to the **printf()** function, except that the variable argument list is replaced by *args*. *args* should be initialized by the **va_start()** macro before the call to **vprintf()** is made.

The return value of **vprintf()** is equal to the number of characters written or is a negative value if an error occurred in output.

SEE ALSO

vfprint(), vsprintf(), printf(), va_start()

vsprintf() write formatted ASCII data to a buffer

TYPE Conversion
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf (char *buf, const char *fmt, va_list args);
```

DESCRIPTION

vsprintf() is equivalent to the **sprintf()** function, except that the variable argument list is replaced by *args*. *args* should be initialized by the **va_start()** macro before the call to **vsprintf()** is made.

The return value of **vsprintf()** is the number of characters written, not including the terminating null characters.

SEE ALSO

vfprintf(), vprintf(), printf(), va_start()

wcstombs() convert a sequence of multibyte characters

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

DESCRIPTION

wcstombs() converts a sequence of codes that correspond to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by *s*, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to **wctomb()**, except that the shift state of the **wctomb()** function is not affected.

No more than *n* bytes will be modified in the array pointed to by *s*. If copying takes place between objects that overlap, the behavior is undefined.

If a code is encountered that does not correspond to a valid multibyte character, **wcstombs()** returns **(size_t)-1**. Otherwise, it returns the number of bytes modified, not including a terminating null character, if any.

wctomb() determine # of bytes in multibyte character

TYPE Miscellaneous
COMPATIBILITY ANSI
LIBRARY c.lib

DECLARATION

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

DESCRIPTION

wctomb() determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar* (including any change in shift state). It stores the multibyte character representation in the array object pointed to by *s* (if *s* is not a NULL pointer). At most MB_CUR_MAX characters are stored. If the value of *wchar* is zero, the **wctomb()** function is left in the initial shift state.

If *s* is a NULL pointer, **wctomb()** returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a NULL pointer, the **wctomb()** function returns -1 if the value of *wchar* does not correspond to a valid multibyte character, or returns the number of bytes that comprise the multibyte character corresponding to the value of *wchar*.

In no case will the value returned be greater than the value of the MB_CUR_MAX macro.

write() write to a file or device using unbuffered I/O

TYPE UNIX I/O
COMPATIBILITY Aztec /UNIX
LIBRARY c.lib

DECLARATION

size_t write (int *fd*, void **buf*, size_t *bufsize*);
size_t _write (int *fd*, woid **buf*, size_t *bufsize*);

DESCRIPTION

write() writes characters to a device or disk which has been previously opened by a call to **open()** or **creat()**. The characters are written to the device or file directly from the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer containing the characters to be written.

bufsize is the number of characters to be written.

If the operation is successful, **write()** returns as its value the number of characters written.

write() must be implemented to support output to any files or devices. If file I/O support is desired, write() must update the file position entry in the device table associated with the relevant file descriptor. It should indicate the byte immediately following the last byte transferred. The file size entry in the device table and file system should also be updated.

If file I/O support is not desired, and file descriptors will always have a fixed interpretation (i.e. '1' will refer to port 1 and '2' will refer to port 2, etc.), positioning and file size information may not have to be maintained

See **open(C)** for more information on: file descriptors, device table, and **write()**'s functionality in Standard I/O.

_write() performs the same as **write()** and is provided for internal library use in case you define a version of **write()** that overrides the library version.

DIAGNOSTICS

If the operation is unsuccessful, **write()** returns -1 and places a code in the global integer **errno**. **errno** is set to EBADF if the file descriptor is invalid. It is set to EINVAL if the file or device associated with the file descriptor is only permits read access.

NOTES

write() and **_write()** are system dependent, and must be specially written for your system.

SEE ALSO

open(), close(), creat(), read()

Chapter 11 - Technical Information

This chapter discusses topics that could not be conveniently discussed elsewhere.

It's divided into the following sections:

- Assembly-language functions;
- C-language interrupt routines.

Assembler Functions

This section discusses assembly-language functions that can be called by, and themselves call, C-language functions.

It first discusses the conventions that such functions must follow, and then discusses the in-line placement of assembler statements within C functions.

C-callable, Assembly-language Functions

A C-callable, assembly-language function must obey the conventions that are described in the following paragraphs.

Names of Global Variables and Functions

By default, the names by which assembly-language modules and C-language modules refer to global variables and functions differ slightly: the assembler name is generated from the C name by prepending an underscore character.

Consider, for example, the following C module:

```
int var;
main()
{
        func(var);
}
```

The names by which an assembler module would by default refer to these global variables and functions are `_var`, `_main`, and `_func`.

You can define an alternate naming convention using the compiler's **-yu** option, as follows:

-yup	Assembly names are derived by prepending an underscore to C names;
-yun	Assembler names are the same as C names;
-yua	Assembler names are derived by appending an underscore to C names.

In the following paragraphs, we assume that assembler names are derived from C names by prepending an underscore.

Global Variables

A C module's global variables are in either the uninitialized data segment or the initialized data segment.

An assembler module can create an uninitialized variable that can be accessed by a C function, using the **global** directive.For example, the following code creates the global variable **_var**, which can be accessed as an array by a C function, and reserves 8 bytes of storage for it.

```
global _var,8
```

A C function that wants to access **_var** could have the following declaration:

```
extern short var[];
```

To create an initialized variable that can be accessed by a C function, an assembler module can use the **public** and **dc** directives. For example, the following code creates the public variable **_ptr** that initially contains a pointer to the symbol **str**, and that can be accessed as a **char** pointer by a C function:

```
        dseg
        public      _ptr
_ptr    dc.l        str
```

To access _ptr, a C function could use the following declaration:

```
extern char *ptr;
```

An assembler module can access global initialized or uninitialized variables that are created in C modules by defining the variables with a public directive within the dseg segment. For example, suppose a C module creates a global, uninitialized short named count and a global, initialized short named total using the statement:

```
short count, total=1;
```

An assembler module can access these variables by using the following directives:

```
dseg
public  _count,_total
```

Names of External Functions and Variables

The compiler translates the name of a function or variable to assembly language by truncating the name to 31 characters and optionally adding an underscore to the name (as defined by the **-yu** option). Thus, to be accessible from C modules or to access C modules, assembler modules must obey this convention.

For example, the following C module calls the function **bmp**, which simply adds **10** to the global short **count**. A C module refers to this function as **bmp**, and an assembler module refers to it as **_bmp**.

```
int count;
main()
{
        bmp();
}
```

An assembler version of **_bmp** could be:

```
        dseg
        public      _count
        cseg
        public      _bmp
_bmp:
        add.w       #10,_count
        rts
        end
```

C Function Calls and Returns

The assembler code generated by the compiler for a C call to another function pushes the arguments onto the stack, in the reverse order in which they were specified in the call's argument list, and then calls the function.

An assembler function returns to a C function caller by issuing a **rts** instruction, and leaving the caller's arguments on the stack. The caller then removes the arguments from the stack.

The registers in which a function returns its value depends on the type of the value:

- Non-floating point values are always returned to **d0**.
- When floating point operations are performed by software, **float**s are returned in **d0**, **double**s in **d0** and **d1**, and **long double**s in the pseudo register **.p0**.
- When floating point operations are performed by a 68881, floating point values are returned in **fp0**.

For example, consider the following assembler function, **_sub**, that takes two short arguments that are passed to it on the stack, subtracts them, and returns the difference as the function value. A C function will refer to this function using the name **sub**.

```
        cseg
        public      _sub
_sub:
        mov         4(sp),d0        ;get first argument
        sub         6(sp),d0        ;subtract second from first
        rts
```

The following C function calls **sub** to subtract **b** from **a**, and stores the difference in **c**:

```
main()
{
        short a,b,c;
        ...
        c = sub(a,b);
}
```

Register Usage

An assembler function that is called by a C function must preserve all registers it uses, except for those that the calling function uses for temporary values.

The registers that a module uses for temporary values are defined when the module is compiled, with the **-yt** option; by default, these are data registers **d0-d1** and address registers **a0-a2**.

Pascal Function Calls and Returns

On entry to a pascal function, the stack contains the following:

- space for the function's return value (only if the function has a return value);
- arguments, which are pushed onto the stack in the order that they appear in the source argument list (i.e. just the reverse of the order for a C function);
- the return address.

If the value that a pascal function is to return is four bytes or less, the value is returned in the space that is reserved for it on the stack. If the return value contains more than four bytes, the function should place a pointer to it in the reserved stack area. Unlike a C function, a pascal function must remove the caller's argument from the stack before returning.

Embedded Assembler Source

Assembler statements can be embedded in a C module by surrounding them with **#asm** and **#endasm** statements. The pound sign (#) must be the first character on the line, and the letters must be lower case.

Embedded assembler code must preserve the contents of all registers it uses, except for those used for temporary values.

It should make no assumptions about the contents of the registers, since the code that the compiler currently generates for C statements may change in the future.

To be safe, a **#asm** statement should be preceded by a semicolon. This avoids problems in which the compiler mistakenly puts a label that is the target of a jump statement after, rather than before, in-line assembly code.

In general, it is safest to contain assembly code in a separate assembler module rather than embedding it in C source.

Interrupt Handlers

An interrupt handler can be written in C, with the following provisos: it must have a small assembly language routine that performs the initial and final processing of an interrupt; and it must restrict its use of the library functions. These provisos are discussed in the following paragraphs.

The Assembly Language Routine

When the assembly language front-end to a C interrupt handler is activated by an interrupt, it must do the following:

- Save on the stack the registers that the C routine uses for holding temporary values;
- If the C routine uses a small memory model, the assembly language routine must save the small memory model support register and set in it the value **__H1_org+32766**. **__H1_org** is a linker-created symbol whose value is the starting address of the interrupt handler's initialized data segment. In case you can't tell, **__H1_org** begins with two underscores, and has one in the middle;
- **jsr** to the C routine.

It's not necessary for the assembly language routine to save other registers (i.e. registers used for holding the C routine's register variables or the frame pointer register); this will automatically be done by the C routine.

The C routine should return in the usual way; i.e. by executing a return instruction or by executing its last instruction. The assembly language routine should then restore all registers that it saved and issue an **rte** instruction.

Here is a sample assembly language routine named **_intbegin**. It saves the default temporary registers **d0-d3** and **a0-a2**; saves and initializes the default small model support register **a5**; and calls the C language routine whose C name is **intfunc**:

```
        public      _intbegin,_intfunc,__H1_org
_intbegin
        movem.l     d0-d3/a0-a2/a5,-(sp)
        move.l      #__H1_org+32766,a5
        jsr         _intfunc
        movem.l     (sp)+,d0-d3/a0-a2/a5
        rte
```

Use of Library Functions By Interrupt Routines

A C interrupt routine can call the reentrant library functions that are provided with Aztec C68k/ROM; it usually shouldn't call the non-reentrant library functions. A function is reentrant if it doesn't access global or static variables, and is non-reentrant if it does.

The non-reentrant library functions are these:

- The high-level buffer-allocation functions **malloc()**, **free()**, etc.
- sprintf();
- sscanf();
- The standard I/O functions, usually;
- The unbuffered I/O functions, usually.

The standard I/O functions are not reentrant, because they have global control blocks and because they call the non-reentrant **malloc()** and **free()** functions. An interrupt routine can call the standard I/O functions if those calls meet certain requirements: the calls can't modify control block fields that may be accessed by the standard I/O calls of an interrupted process, and they can't call **malloc()** or **free()**. For example, an interrupt routine can issue standard I/O calls to pre-opened streams whose standard I/O operations are unbuffered. It can also issue standard I/O calls to pre-opened buffered streams, if the buffer has been preallocated, and if it only accesses those streams.

The unbuffered I/O functions (which you must write) are usually not reentrant, because they usually have a global table. But an interrupt routine can call the unbuffered I/O functions if those calls don't modify fields that may be accessed by the calls of an interrupted process.

ERROR MESSAGES 12

Chapter 12 - Error Messages

This chapter discusses error messages that can be generated by the Aztec compiler, assembler, and linker. It is divided into three sections.

Each section offers a summarized list of error messages followed by the detailed explanations of each error message

Compiler Error Messages

Compiler error messages are broken down into three groups. If the error message displayed does not give you a number before the message, refer to the sections "Fatal Compiler Error Messages" and "Compiler Internal Errors" which follow the numbered messages.

Note:

- Error codes greater than 200 will occur only if there is something wrong with the compiler. If you get such an error, please send us the program that generated the error.

```
1: bad digit in octal constant
2: string space exhausted
3: unterminated string
4: argument type mismatch
5: invalid type for function
6: inappropriate arguments
7: bad declaration syntax
8: syntax error in typecast
9: invalid operand of & (address of)
10: array size must be positive integer
11: data type too complex
12: invalid pointer reference
13: unimplemented type
14: long switches not supported
15: storage class conflict
16: data type conflict
17: internal
18: data type conflict
19: bad syntax
20: structure redeclaration
21: missing }
22: syntax error in structure declaration
23: syntax error in enum declaration
24: need right parenthesis or comma in arg list
25: structure member name expected here
26: must be structure/union member
27: invalid typecast
28: incompatible structures
29: invalid use of structure
30: missing : in ? conditional expression
31: call of non-function
32: invalid pointer calculation
```

33: invalid type
34: undefined symbol
35: typedef not allowed here
36: no more expression space
37: invalid or missing expression
38: no auto. aggregate initialization allowed
39: enum redeclaration
40: internal [see error 17]
41: initializer not a constant
42: too many initializers
43: initialization of undefined structure
44: missing right paren in declaration
45: bad declaration syntax
46: missing closing brace
47: open failure on include file
48: invalid symbol name
49: multiply defined symbol
50: missing bracket
51: lvalue required
52: too many right paren's
53: multiply defined label
54: too many labels
55: missing quote
56: missing apostrophe
57: line too long
58: invalid # encountered
59: macro too long
60: loss of const/volatile info
61: reference to undefined structure
62: function body must be compound statement
63: undefined label
64: inappropriate arguments
65: invalid function argument
66: expected comma
67: invalid else
68: bad statement syntax
69: missing semicolon
70: goto needs a label
71: statement syntax error in do-while
72: statement syntax error in for
73: statement syntax error in for body
74: expression must be integer constant

```
75: missing colon on case
76: too many cases in switch
77: case outside of switch
78: missing colon on default
79: duplicate default
80: default outside of switch
81: break/continue error
82: invalid character
83: too many nested includes
84: constant expression expected
85: not an argument
86: null dimension in array
87: invalid character constant
88: not a structure
89: invalid use of register storage class
90: symbol redeclared
91: invalid use of floating point type
92: invalid type conversion
93: invalid expression type for switch
94: invalid identifier in macro definition
95: obsolete
96: missing argument to macro
97: too many arguments in macro definition
98: not enough arguments in macro reference
99: internal [see error 17]
100: internal [see error 17]
101: missing close parenthesis on macro reference
102: macro arguments too long
103: #else with no #if
104: #endif with no #if
105: #endasm with no #asm
106: #asm within #asm block
107: missing #endif
108: missing #endasm
109: #if value must be integer constant
110: invalid use of : operator
111: invalid use of a void expression
112: invalid use of function pointer
113: duplicate case in switch
114: macro redefined
115: keyword redefined
116: field width must be > 0
```

```
117: invalid 0 length field
118: field is too wide
119: field not allowed here
120: invalid type for field
121: ptr/int conversion
122: ptr & int not same size
123: far/huge ptr & ptr not same size
124: invalid ptr/ptr expression
125: too many subscripts or indirection on integer
126: too many arguments
127: too few arguments
128: #error
129: #elif with no #if
130: obsolete
131: ## at the beginning/end of macro body
132: obsolete
133: # not followed by a parameter
134: name of macro parameter was not unique
135: attempt to undefine a predefined macro
136: invalid #include directive
137: macro buffer overflowed
138: missing right paren
139: missing identifier
140: obsolete
141: invalid character
142: range-modifier ignored
143: range-modifier syntax error
144: invalid operand for sizeof
145: function called without prototype
146: constant value too large
147: invalid hexadecimal constant
148: invalid floating constant
149: invalid character on control line
150: unterminated comment
151: no block level extern initialization
152: missing identifier in parameter list
153: missing static function definition
154: function definition can't be via typedef
155: file must contain external definition
156: wide string literal not allowed here
157: incompatible function declarations
158: called function may not return incomplete type
```

```
159: syntax error in #pragma
160: auto variable not used in function
161: function defined without prototype
162: can't take address of register class
163: upper bits of hex character constant ignored
164: non-void type function must have return value
165: struct/union must be declared outside of
     prototype
166: enum must be declared outside prototype
167: expression too complex - use -MR option
168: invalid type for arg of regcall-type function
169: function declaration incompatible with prior use
170: invalid register for registerized argument
171: %%arg not found
172: invalid return type for regcall-type function
```

Fatal Compiler Error Messages

In the following "%%" is used to indicate a variable number of items can appear at that place.

```
Can't open file '%%' for input!
Cannot create output file '%%'!
Dump file not found!
Error creating dump!!
Error reading dump file!
Illegal '-%%' option: %%
Illegal '-h' option: '%%'
Illegal 3.6 '+x' option: '%%'
Illegal 3.6 option: '%%'
Illegal option: '%%'
Invalid register in -YF option
Invalid register in -YD option
More than one output filename specified
Multiple '-h' options specified
Multiple input files specified!
Need valid register here
No input file was specified!
Out of disk space
Out of memory!
Outbyte of 0x%%
Pre-compiled header not in proper format!
```

```
Pre-compiled header uses wrong size int!
Premature end of file in macro definition
Redefining label type
Symbol required
Too few arguments for '%c' option!
Too many -i options
Unable to execute as68
```

Compiler Internal Errors

```
Attempt to use expression tree entry twice
Attempt to release item already free
Bad arginfo
Bad cast - 'cast'
Bad op in cgen: 'op'
Bad operator in donode!
Bad value in outarg: opr
Bf_shift didn't work
Optim failure
Out of registers!
```

Explanations of Compiler Error Messages

Note:

- Error codes greater than 200 will occur only if there is something wrong with the compiler. If you get such an error, please send us the program that generated the error.

`1: bad digit in octal constant`

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., **`0x1, 0xAA0, 0xFFF`**).

`2: string space exhausted`

This error should not occur.

`3: unterminated string`

All strings must begin and end with double quotes ("). This message indicates that a double quote has remained unpaired.

`4: argument type mismatch`

This warning is given if the argument specified in a function call does not match that of the function's prototype. Although the warning is given, the argument will be converted to the appropriate type before being passed. To avoid the warning, the argument can be preceded by a type cast to the appropriate type.

`5: invalid type for function`

Functions may be declared to return any scalar type as well as certain aggregate types such as structures. Functions are not allowed to return arrays. All definitions or declarations of a function or a function pointer that return an array will generate this error message. For example:

```
char(* f)()[];
```

`6: inappropriate arguments`

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to **int**, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
shrt arg 1;                /* mispelled or invalid keyword */
double arg 2;
{                          /* function body */
}

goodfunction(arg1,arg2)
float arg1;
int arg2;                  /* this line is not required */
{                          /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;                  /* correct  */
char c d;                  /* error 7 */
char *s1, *s2              /* error 7 detected here  */
float k;
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a **#include**'d file will be detected back in the file being compiled or in another **#include** file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in typecast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number);          /* incorrect usage */
i = 3 * ((int)number);         /* correct usage *
```

9: invalid operand of & (address of)

This error is given if the program attempts to take the address of something that does not have an address associated with it.

```
#define FOUR 4
char *addr;

addr = &FOUR;        /* error 9, can't take the address of a constant */
```

10: array size must be positive integer

The dimension of an array must be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];                    /* meaningless */
extern char goodarray[];             /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, **goodarray** is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
...
```

11: data type too complex

This message is best explained by example:

```
char *******foo;
```

The form of this declaration implies six pointers-to-pointers. The seventh asterisk indicates a pointer to a **char**. The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error. However it is to be hoped that such a construct will never be needed.

12: invalid pointer reference

This error message will occur if pointer indirection is attempted on a type which cannot physically represent a printer value. The only types, other than printers themselves, which can hold printer values, are **int**, **short**, and **long** (as well as their unsigned counterparts). All other C types will generate this error. For example,

```
char c;
*c = 5;
```

will generate this error.

13: unimplemented type

This error should not occur with the current compiler since all the ANSI specified data types are supported. In previous versions of the compiler, the **enum** keyword was allowed but the type was not supported.

`14: long switches not supported`

This error should not occur with versions 5.0 and higher, since long switches are supported.

`15: storage class conflict`

Only automatic variables and function parameters can be specified as **register**.

This error can be caused by declaring a **static register** variable. While structure members cannot be given a storage class at all, function arguments can be specified only as **register**.

A **register int** *i* declaration is not allowed outside a function—it will generate error 89 (see below).

`16: data type conflict`

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say **`long int i`**, and **`unsigned int j`**, it is meaningless to use **`double int k`** or **`float char c`**. In this respect, the compiler checks to make sure that **int**, **char**, **float** and **double** are used correctly.

Data	Type	Interpretation
char	character	1
int	integer	2
unsigned/ unsigned int	unsigned integer	2
short	integer	2
long/ long integer	long integer	4
float	floating point number	4
long float/ double	double precision float	8

`17: internal`

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of Manx, as it could be a bug in the compiler.

18: data type conflict

This message indicates an error in the use of the **long** or **unsigned** data type. **long** can be applied as a qualifier to **int** and **float**. **unsigned** can be used with **char**, **int** and **long**.

```
long i;                    /* a long int */
long float d;              /* a double */
unsigned u;                /* an unsigned int */
unsigned char c;
unsigned long l;
unsigned float f;          /* error 18 */
```

19: bad syntax

This error occurs if the **#line** preprocessor directive is followed by something other than a numeric constant or macro that expands to one.

```
#line 100 "filename"              /* correct */
#line "filename"                  /* error 19 */
```

20: structure redeclaration

This message informs you that you have tried to redefine a structure.

21: missing }

The compiler requires a comma after each member in the list of fields for a structure initialization. After the last field, it expects a right (close) brace.

For example, this program fragment will generate error 21, since the initialization of the structure named **emily** does not have a closing brace:

```
struct john {
   int bone;
   char license[10];
} emily = {
   1,
   "23-4-1984";
```

22: syntax error in structure declaration

This error occurs in a structure declaration that is missing the opening curly brace or when the left curly brace is followed by a right curly brace with nothing but white space.

```
struct                  /* error 22, missing left curly brace */
   int a;
   long b;
}
```

23: syntax error in enum declaration

This error occurs in an **enum** specification that is missing the opening curly brace or when the left curly brace is followed by a right curly brace with nothing but white space.

```
enum colors {
}                    /* error 23, nothing in enumerator list */
```

24: need right parenthesis or comma in arg list

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that **getchar** is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funccall(arg1, arg2, arg3);
```

25: structure member name expected here

The symbol name following the dot operator or the arrow must be valid. A valid name is a string of alphanumerics and underscores. It must begin with an alphabetic (a letter of the alphabet or an underscore). In the last line of the following example, **(salary)** is not valid because '(' is not an alphanumeric.

```
empptr = &anderson;
empptr->salary = 12000;              /* these three lines */
(*empptr).salary = 12000;            /* are */
anderson.salary = 12000;             /* equivalent */
empptr = &anderson.;                 /* error 25 */
empptr- = 12000;                     /* error 25 */
anderson.(salary) = 12000;           /* error 25 */
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the -s option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.

27: invalid typecast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure david { ... } amy;
amy = (struct david)(expression );          /*error 27*/
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. Both structures must have the same structure tag. For example:

```
struct david emily;
struct david amy;
 ...
emily = amy;
```

29: invalid use of structure

Not all operators can accept a structure as an operand.Also, structures cannot be passed as arguments.However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

expression ? *statement1* : *statement2*

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

Error 31 is generated by an expression that attempts to call a data item. The following code will generate an error 31:

```
int a;
a();
```

Error 31 is often caused by an expression that is missing an operator. For example, Error 31 will be generated if the expression `a * (b+c)` is coded `a (b+c)`.

32: invalid pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: invalid type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct joey , alice;
a = -array;
b = -alice;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of **sizeof**(*expression*) and the cast operator. Compare the accompanying examples:

```
struct lucille {
  int i;
}andrew;
typedef double bigfloat;
typedef struct lucille foo;
j =4 * bigfloat f;                    /* error 35 */
k = &foo;                             /* error 35 */
x= sizeof(bigfloat);
y= sizeof(foo);                       /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as

andrew). It is no more meaningful to take the address of a structure type than any other data type, as in **&int**.

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the **-e** option to increase the number of available entries in the expression table. For more information, see the **Compiler** chapter.

37: invalid or missing expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (—), unary minus (-), typecast, pointer-to (*), address-of (&), and **sizeof**.

38: no auto. aggregate initialization allowed

This error should not occur.

39: enum redeclaration

This error occurs when an **enum** identifier is used more than once in defining the value of enumeration constants.

```
enum states { NY, CA, PA }
enum states { IL, FL, NJ }      /* error 39, states has already been
used */
```

40: internal [see error 17]

41: initializer not a constant

In certain initializations, the expression to the right of the equal sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actually occurs at link-time, and not at run-time.

```
{
  int i = 3;
  static int j = (2 + i);              /* illegal */
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
        struct {
                struct {
                        char array[];
                } substruct;
        } superstruct =
version 1:
        {
                "aBDdefghij"
        };
  version 2:
        {
                {
                        { 'a','b','c',...,'i','j'}
                }
        };
```

In **version 1**, the initializers are copied byte-for-byte onto the structure, **superstruct**.

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[10] = "abcdefghij";
```

This will generate error 42 because the string constant on the right is null-terminated. The null terminator ('\0' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43: initialization of undefined structure

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct david {...};
struct dog david = { 1, 2, 3};          /* error 43 */
```

44: missing right paren in declaration

This error occurs in the declaration of a function pointer when the right parenthesis is left out.

```
int (* fp)();                           /* error 44 */
int (* fp();                            /* error 44 */
```

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the **while** loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the **while** loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
main()
{
   int i, j;
   char array[80];
   gets(array);
   i = 0;
   while (array[i]) {
         putchar(array[i]);
         i++;
   for ( i=0; array[i];i++) {
         for (j=i + 1; array[j]; j++) {
               printf("elements %d and %d are ",i, j);
               if (array[i] == array[j])
                     printf("the same\n");
               else
                     printf("different\n");
         }
         putchar('\n');
   }
}
```

47: open failure on include file

When a file is #included, the compiler will look for it in a default area (see the **Compiler** chapter). This message will be generated if the file could not be opened. An open failure usually occurs

when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in an include statement, but this diminishes flexibility somewhat.

48: invalid symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The succeeding characters may be any combination of alphanumerics (alphabetics and numerals). The following symbols will produce this error code:

```
2nd_time,
dont_do_this!
```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```
int i, j, k, i;                    /* illegal */
```

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only lvalues are allowed to stand on the left-hand side of an assignment. For example:

```
int num;
num = 7;
```

They are distinguished from rvalues, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An lvalue may be thought of as a bucket into which an rvalue can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue, `y`, be assigned to another lvalue, `x`:

```
#define NUMBER 512
x = y;
1024 = z;                    /* wrong;/rvalues are reversed */
NUMBER = x;                  /* wrong; NUMBER is  an rvalue */
```

Some operators which require lvalues as operands are increment (++), decrement (—), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
register int i, j;
i = 3;
j = &i;
```

52: too many right paren's

This error should never be returned from this compiler.

53:multiply defined label

On occasions when the **goto** statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. although this table is fixed in size, it should satisfy the requirements os any reasonable C program. C was structured to discourage extravagance in the use of **gotos**. Strictly speaking, **goto** statements are not required by any procedure in C; they are primarily recommended as a quick and simple means of exiting from a nested structure.

This error indicates that you should significantly reduce the number of **gotos** in your program.

55: missing quote

The compiler found a mismatched double quote (") in a **#define** preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"
"this is a string with an embedded quote: \". "
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a **#define** preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\'';              /* c is initialized to single quote */
```

`57: line too long`

Lines are restricted in length by the size of the buffer used to hold them. This restriction varies from system to system. However, logical lines can be infinitely long by continuing a line with a backslash-newline sequence. These characters will be ignored.

`58: invalid # encountered`

The pound sign (#) begins each command for the preprocessor: **#include, #define, #if, #ifdef, #ifndef, #else, #endif, #asm, #endasm, #line and #undef**. These symbols are strictly defined. The pound sign (#) must be in column one and lower case letters are required.

`59: macro too long`

Macros can be defined with a preprocessor command of the following form:

#define [*identifier*] [*substitution text*]

The compiler then proceeds to replace all instances of *identifier* with the *substitution text* that was specified by the **#define**.

This error code refers to the *substitution text* of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (\), for practical purposes the size of a macro has been limited to 255 characters.

`60: loss of const/volatile info`

This error occurs when passing the address of a variable that is declared as **const** and/or **volatile**.

```
extern const volatile int clock_time;
set_time(&clock_time);                    /* error 60 */
```

`61: reference to undefined structure`

This message comes in two forms:

1) As a warning, due to referencing an undefined structure member.

2) As an error, when trying to obtain the size of an undefined structure.

```
a = sizeof(struct nodef);        /* error 61 unless nodef
                                            has been defined */
```

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()
{
   return 1;
}
```

This error can also be caused by an error inside a function declaration list, as in:

```
func(a, b)
int a; chr b;
{
   ...
}
```

63: undefined label

A **goto** statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to go to a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding **goto**, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function(string)
char *string;
{
        char *func1();                  /* correct */
        double func2(x,y);              /* wrong */
   ...
}
```

In this example, `function()` is being defined, but `func1()` and `func2()` are being declared.

65: invalid function argument

This error occurs in a function definition that contains an argument that is not a valid identifier.

```
sub(a, 2b) {        /* error 65 because identifiers can't begin
                                    with a numeric character */
```

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An **else** was found which is not associated with an **if** statement. **else** is bound to the nearest **if** at its own level of nesting. So **if-else** pairings are determined by their relative placement in the code and their grouping by braces.

```
if(...) {
        ...
   if (...) {
        ...
   } else if (...)
        ...
   } else {
        ...
   }
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the **if** and **else-if** means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding **if** statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the **else** statement. As shown here, the **else** is paired with the first **if**, not the second.

68: bad statement syntax

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, condi-

tional or function. Once the compiler has reached a non-declaration, a keyword such as **char** or **int** must not lead a statement; compare the use of the casting operator:

```
func()
{
   int i;
   char array[12];
   float k = 2.03;
   i = 0;
   int m;                                  /* error 68 */
   j = i + 5;
   i = (int) k;                            /* correct */
   if (i) {
         int i = 3;
         j = i;
         printf("%d",i);
   }
   printf("%d%d\n",i,j);
}
```

This trivial function prints the values **3**, **2** and **3**. The variable **i** which is declared in the body of the conditional **if** lives only until the next right brace; then it dies, and the original **i** regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is similar to error code 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: goto needs a label

Compare your use of **goto** with this example. This message says that you did not specify where you wanted to **goto** with **label**:

```
   goto label;
   ...
label:
   ...
```

It is not possible to **goto** just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in do-while

The body of a **do-while** may consist of one statement or several statements enclosed in braces. A **while** conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, do not forget the **while** conditional.

72: statement syntax error in for

This error occurs when the first of the two semicolons that separate the three expressions found in a **for** loop condition are missing.

```
for (i=0 i; i++) {          /* error 72 due to missing semicolon */
```

73: statement syntax error in for body

This error occurs when the second of the two semicolons that separate the three expressions found in a **for** loop condition is missing.

```
for (i=0; i i++) {          /* error 73 due to missing semicolon */
```

74: expression must be integer constant

This error occurs when a variable occurs instead of an integer constant in declaring the size of an array, initializing an element in an **enum** list, or specifying a **case** constant for a **switch**.

75: missing colon on case

This should be straightforward. If the compiler accepts a **case** value, a colon should follow it. A semi-colon must not be accidently entered in its place.

76: too many cases in switch

The compiler reserves a limited number of spaces in an internal table for **case** statements. If a program requires more cases than the table initially allows, it becomes necessary to tell the compiler what the table value should be changed to. It is not necessary to know exactly how many are needed; an approximation is sufficient, depending on the requirements of the situation.

77: case outside of switch

The keyword, **case**, belongs to just one syntactic structure, the **switch**. If **case** appears outside the braces which contain a **switch** statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon on default

This message indicates that a colon is missing after the keyword, **default**. Compare error 75.

79: duplicate default

The compiler has found more than one **default** in a **switch**. **switch** will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a **case** statement. It is analogous to the **else** companion to the conditional, **if**. Just as there is one **else** for every **if**, only one default **case** is allowed in a **switch** statement. However, unlike the **else** statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, **default**, is used just like **case**. It must appear within the brackets which delimit the switch statement.

81: break/continue error

break and **continue** are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a **switch** statement. But when the keywords, **break** or **continue**, are used outside of these contexts, this message results.

82: invalid character

Some characters simply do not make sense in a C program, such as **$** and **@**. Others, for instance the pound sign (**#**), may be valid only in particular contexts.

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, **file D** is not allowed to have a **#include** in the compilation of **file A**.

file A	file B	**file C**	**file D**
#include B	**#include C**	**#include "D"**	

84: constant expression expected

This error occurs when an integer constant is missing, such as in initializing an element in an **enum** list, specifying a **case** constant for a **switch**, or for a **#if** preprocessor directive.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an **extern** declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
extern char array[][12];                /* correct */
extern char badarray[5][];              /* wrong */
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as **'a'** or **'ab'**. There is no analog to a null string, so '' (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (**\b, \n, \t** etc.) are singular, so that the following are valid: **'\n', '\na', 'a\n'**. **'aaa'** is invalid.

88: not a structure

Occurs only under compilation without the **-s** option. A name used as a structure does not refer to a structure, but to some other data type.

```
int i;
i.member = 3;                       /* error 88 */
```

89: invalid use of register storage class

A globally defined variable cannot be specified as a register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: invalid use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: invalid type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;
float j;
char *ptr;
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type **char** and **short** become **int**, and **float** becomes **double**. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a **float** will evaluate to a **double**.

Types have the following hierarchy:

double float
long
unsigned
int short, char

This error can also be caused by an attempt to return a structure, since the structure is being cast to the type of the function, as in:

```
int func()
{
   struct tag sam;
   return sam;
}
```

93: invalid expression type for switch

Only a **char**, **int** or **unsigned** variable can be switched. See the example for error 74.

94: invalid identifier in macro definition

This error occurs in a macro definition that contains one or more arguments that are not valid *identifiers*.

```
#define add(a,2b) (a+2b)   /* error 94 because identifiers can't begin
                                        with a numeric character */
```

95: obsolete

Error codes interpreted as obsolete do not occur in the current version of the compiler. Some simply no longer apply due to the increased adaptability of the compiler. Other error codes have been

translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact Manx for information.

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

```
#define reverse(x,y,z) (z,y,x)
func(reverse(i,,k));
```

97: too many arguments in macro definition

This error occurs in a macro definition that contains more than 32 arguments in its definition.

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define exchange(x,y) (y,x)
func(exchange(i));                /* error 98 */
```

99: internal [see error 17]

100: internal [see error 17]

101: missing close parenthesis on macro reference

A right (closing) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between **#if** and **#else** is analogous to that which exists between the control flow statements, **if** and **else**. Obviously, much depends upon the relative placement of the state-

ments in the code. However, **#if** blocks must always be terminated by **#endif**, and the **#else** statement must be included in the block of the **#if** with which it is associated. For example:

```
#if ERROR 0
  printf("there was an error\n");
#else
  printf("no error this time\n");
#endif
```

#if statements can be nested, as below. The range of each **#if** is determined by a **#endif**. This also excludes **#else** from **#if** blocks to which it does not belong:

```
#ifdef  JAN1
  printf("happy new year!\n");
#if    sick
  printf("i think i'll go home now\n");
#else
  printf("i think i'll have another\n");
#endif
#else
  printf("i wonder what day it is\n");
#endif
```

If the first **#endif** was missing, error 103 would result. And without the second **#endif**, the compiler would generate error 107.

104: #endif with no #if

#endif is paired with the nearest **#if**, **#ifdef** or **#ifndef** which precedes it. (See error 103.)

105: #endasm with no #asm

#endasm must appear after an associated **#asm**. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a **#endasm** without having found a previous **#asm**. If the **#asm** was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the **#asm** keyword must not appear between a paired **#asm**/**#endasm**. When a piece of in-line assembly is aug-

mented for temporary purposes, the old **#asm** and **#endasm** can be enclosed in comments as place-holders.

```
#asm
        /* temporary asm code */
        /*  #asm old beginning */
        /* more asm code */
#endasm
```

107: missing #endif

A **#endif** is required for every **#if**, **#ifdef** and **#ifndef**, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first **#endif**. Backtrack to the previous **#if** and form the pair. Assign the next **#endif** with the nearest unpaired **#if**. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a **#endasm** in all cases. **#asm** must always be paired with a **#endasm**.

109: #if value must be integer constant

#if requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers, then

```
#if DIFF = 'A'-'a'
#if (WORD &= ~MASK) > 8
#if MAR | APR | MAY
```

are all legal expressions for use with **#if**.

110: invalid use of : operator

The colon operator occurs in two places:

- Following a question mark as part of a conditional, as in

```
(flag ? 1 : 0);
```

- Following a label inserted by the programmer or following one of the reserved labels, **case** and **default**.

111: invalid use of a void expression

This error can be caused by assigning a **void** expression to a variable, as in this example:

```
void func();
int h;
h = func(arg);
```

112: invalid use of function pointer

For example,

```
int (*funcptr) ();
...
funcptr++;
```

funcptr is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not subject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

A **switch** statement has two **case** values which are the same. Either the two **cases** must be combined into one, or one must be discarded. For instance:

```
switch (c) {
case NOOP:
   return (0);
case MULT:
   return (x * y);
case DIV:
   return (x / y);
case NOOP:
default:
   return;
}
```

The **case** of **NOOP** is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define  islow(n) (n>=0&&n<5)
...
#define  islow(n) (n>=0&&n<=5)
```

The macro, **islow**, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define islow(n) n>0&&n>=<5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define NULL 0
...
#define NULL 0
```

But these are different from:

```
#define NULL '\0'
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define int foo
```

If you have a variable which may be either, for instance, a **short** or a **long** integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef LONGINT
  long i;
#else
  short i;
# endif
```

Another possibility is through a **typedef**:

```
#ifdef LONGINT
  typedef  long   VARTYPE;
#else
  typedef  short   VARTYPE;
#endif
VARTYPE i;
```

116: field width must be > 0

A field in a bit field structure can not have a negative number of bits.

117: invalid 0 length field

A field in a bit field structure can not have zero bits.

118: field is too wide

A field in a bit field structure can not have more than 16 bits.

119: field not allowed here

A bit field definition can only be contained in a structure.

120: invalid type for field

The type of a bit field can only be of type **int** or **unsigned int**.

121: ptr/int conversion

The compiler issues this warning message if it must implicitly convert the type of an expression from pointer to **int** or **long**, or vice versa.

If the program explicitly casts a pointer to an **int** this message will not be issued. However, in this case, error 122 may occur.

For example, the following will generate warning 121:

```
char *cp;
int i;
...
i=cp;                /* implicit conversion of char to int */
```

When the compiler issues warning 121, it will generate correct code if the sizes of the two items are the same.

122: ptr & int not same size

If a program explicitly casts a pointer to an **int**, and the sizes of the two items differ, the compiler will issue this warning message. The code that is generated when the converted pointer is used in an expression will use only as much of the least significant part of the pointer as will fit in an **int**.

123: far/huge ptr & ptr not same size

This error occurs when trying to assign a near pointer to a **far** or **huge** pointer. A warning is generated when casting a **far** or **huge** pointer to a near pointer.

124: invalid ptr/ptr expression

If a program attempts to assign one pointer to another without explicitly casting the two pointers to be of the same type, and the types of the two pointers are in fact different, the compiler will issue this warning message.

The compiler will generate code for the assignment, and if the sizes of the two pointers are the same, the code will be correct. But if the sizes differ, the code may not be correct.

125: too many subscripts or indirection on integer

This warning message is issued if a program attempts to use an integer as a pointer; that is, as the operand of a star operator.

If the sizes of a pointer and an **int** are the same, the generated code will access the correct memory location, but if they are not, it will not.

For example,

```
char c;
long g;
*0x5c=0;            /* warning 125, because 0x5c is an int */
c[i]=0;             /* warning 125, because c+i is an int */
g[i]=0;             /* error 12, because g+i is along */
```

126: too many arguments

This error occurs when a function is invoked with more arguments than is specified in its prototype or definition. The only exception allowed is when a variable number of arguments is specified in the prototype.

127: too few arguments

This error occurs when a function is invoked with less arguments than is specified in its prototype or definition.

128: #error

This error is generated by the **#error** directive and is followed by the optional sequence of preprocessing tokens found in the source code.

129: #elif with no #if

This error occurs when the **#elif** preprocessor directive is used without a preceding **#if** directive.

130: obsolete [see error 95]

131: ## at the beginning/end of macro body

This error occurs when a **##** is found as the first or last end of a macro definition body.

```
#define TWOSHARP 2##              /* error 131 */
```

132: obsolete[see error 95]

133: # not followed by a parameter

This error may occur in a **#define** macro in which the # operator is applied to a parameter in the replacement list. If the # token is not followed by a parameter, this error message will be generated.

134: name of macro parameter was not unique

This error should never be returned from this compiler.

135: attempt to undefine a predefined macro

This error occurs when attempting to use a **#undef** on those macros that are predefined by the compiler, such as __STDC__, __TIME__, __DATE__, __FILE__, __LINE__, and __FUNC__.

```
#undef __TIME__                   /* error 135 */
```

136: invalid #include directive

This error occurs when the **#include** directive is not followed by a string literal or a *filename* enclosed in < > signs.

```
#include filename                  /* error 136 */
```

137: macro buffer overflowed

This error should never be returned from this compiler.

138: missing right paren

This error occurs when attempting to use the defined directive with a left parenthesis and no matching right parenthesis.

```
#if defined(MACINTOSH              /* error 138 */
```

139: missing identifier

This error occurs when attempting to use the defined directive with no identifier following the defined keyword.

```
#if defined                        /* error 139 */
```

140: obsolete

[see error 95]

141: invalid character

This error should never be returned from this compiler.

142: range-modifier ignored

Using a range modifier (**near, far,** etc.) on a structure or union member is allowed by the parser but has no effect and is ignored.

Note: This error should never be returned from this compiler.

143: range-modifier syntax error

A range-modifier is illegal as part of a function declaration. You cannot say that a function is **near, far, huge,** etc.

Note: This error should never be returned from this compiler.

144: invalid operand for sizeof

This error occurs when attempting to obtain the **sizeof** of something other than a previously defined data structure.

```
sub()
{
   int i = sizeof(sub);            /* error 144 */
}
```

145: function called without prototype

This warning is generated for functions that are called without having been prototyped. It only occurs when compiling with the **-wp** option.

146: constant value too large

This error occurs when attempting to use a constant larger than the unsigned long **0xffffffff** in an expression.

147: invalid hexadecimal constant

This error occurs when the character following a 0x or 0X is not a valid hexadecimal constant.

```
int i = 0xg2;          /* error 147, 'g' is not a valid hex constant */
```

148: invalid floating constant

This error occurs if the first letter excluding the optional sign following the **e** or **E** in a floating point number is something other than a digit.

```
double d = 123e+f;   /* error 148, 'f' is not a digit */
```

149: invalid character on control line

This error occurs on conditional preprocessor lines that expect a single constant expression but get extra information.

```
#if CONST invalid    /* error 149 due to extra characters "invalid" */
```

150: unterminated comment

This error occurs if the start of a comment (/*) is not terminated with (*/) before the end of the file.

151: no block level extern initialization

This error occurs when initialization of an **extern** variable is attempted inside a function. Initialization of **externs** is permissible outside of functions, or the function can declare the variable as **extern** and then initialize it further down in the code using an assignment statement.

152: missing identifier in parameter list

This error occurs when the type of an argument is specified in a function definition without being followed by the argument itself.

```
sub(int ) {          /* error 152, missing the name of the int argument */
```

153: missing static function definition

This error occurs if a function has been declared as static in a file and has not been followed by its actual definition further down in the file.

154: function definition can't be via typedef

This error occurs when incorrectly defining a function using a **typedef**. It is possible to define a **typedef** that is a function such as:

```
typedef int F(void);
```

which sets the type **F** to be a function with no arguments returning **int**. Then, a function can be declared such as:

```
F f;
```

which IS legal. However, the function definition:

```
F f {}
```

is illegal.

155: file must contain external definition

This error occurs in a file with no external data or function definitions when compiling with the **-pa** option to use the ANSI preprocessor.

156: wide string literal not allowed here

This error occurs when attempting to use a wide string literal with the **#include** or **#line** directives.

```
#include L"filename"              /* error 156 */
```

157: incompatible function declarations

This error occurs if a function declaration does not match a previous definition or declaration for the same function.

158: called function may not return incomplete type

This error occurs when a function attempts to return a structure which has not been defined. If a function is called that returns a structure, but the size of the structure is unknown, then it is not possible for the compiler to know how much data is being returned by the function and how much space to reserve for the return value.

For example:

```
struct foo x();
main()
{
   x();
}
```

159: syntax error in #pragma

This error occurs if the **#pragma** is used for a function call and does not match the following syntax.

```
#pragma regcall([return=]
   func(arg1,arg2,...,argn))
#pragma amicall(base,
   offset,func(arg1,arg2,...,argn))
#pragma libcall func base offset regmask
#pragma syscall func offset regmask
```

160: auto variable not used in function

This warning occurs when compiling with the **-wu** option and a function containing a local variable has not been used.

161: function defined without prototype

This warning occurs:

- when compiling with the **-wp** option and a function does not have its arguments prototyped

or

- if there are no arguments but you have not specified **void**.

162: can't take address of register class

This error occurs when attempting to take the address of a variable that has been declared as a register class variable.

```
register int a;
int *ip;

cp = &a;              /* error 162 */
```

163: upper bits of hex character constant ignored

This warning occurs if the compiler encounters a hexadecimal character constant (specified by **\x**) whose value cannot fit within a single byte. For example:

```
char *cptr = "\x9b7";
```

will generate a warning because **"\x9b7"** cannot be stored within one byte. The compiler will ignore the most significant bits, and use the least significant bits. In the example the compiler will treat **"\x9b7"** as **"\xb7"**. This warning will occur if you accidentally place a digit from **0** through **9** or a letter from **a** through **f** immediately after a **\x** escape sequence. In the example if you intended to have **0x9b** followed by the ASCII digit **7**, you could use string concatenation to produce the desired result:

```
char *cptr = "\x98" "7";
```

164: non-void type function must have return value

This error message can occur only if the **-wr** compiler option is used. If the compiler encounters a function which is defined as returning a value (**int**, **char**, or the like) but which does not have an explicit return. For example:

```
int func()
{
      printf("hello \n");
}
```

would generate this message. Replacing **int func()** with **func()** will not correct the error. The specification **void func()** will correct the problem. The specification of an explicit **return** will also.

165: struct/union must be declared outside of prototype

This warning message will be generated if a **union** or **struct** appears as an argument in a function prototype and there is no previous declaration for the **union** or **struct**. This warning will be generated it there is no **union** or **struct** declaration or if the **union** or **struct** declaration occurs

after the prototype statement. The sequence **struct** declaration, prototype statement, function definition will correct the problem as in this example:

```
struct astruct {
        inta;
        char c;
}

void func(struct astruct arg);

void func(struct astruct arg);
{
   ...
}
```

This problem presents special difficulties when it arises because the intended **union** or **struct** declaration occurs after the prototype definition. A strict interpretation of ANSI rules, in this case, produces results that may seem arbitrary and illogical. Positioning the **union** or **struct** declaration so that it occurs before the prototype definition corrects the problem. If the problem is not corrected, it is unlikely that the program will run correctly.

`166: enum must be declared outside prototype`

This warning message is generated if an **enum** argument appears in a function prototype before the **enum** is defined. To correct the problem, first define the **enum**, next include the function prototype, and then the function definition.

`167: expression too complex - use -MR option`

This error indicates that the temporary registers were exhausted in attempting to generate code for a particular expression. To resolve the problem, either reduce the complexity of the expression by splitting it into two or more statements, or compile with the **-mr** option which increases the number of temporary registers by decreasing the number of register variables allowed.

`168: invalid type for arg of regcall-type function`

This error indicates that the prototype for a 'regcall'-type function (i.e. a function that is specified in a regcall or related pragma and that is passed arguments in register instead of on the stack) defines arguments to be one of the following disallowed types:

- A structure;
- A floating point argument that won't fit in a 32-bit register.

`169: function declaration incompatible with prior use`

This error indicates that the declaration for a function specifies a return type that is compatible with a previous definition (explicit or implicit) of the same function. For example, the compiler will report error 169 for the following:

```
int f(void);
long f(void){}
```

`170: invalid register for registerized argument`

This error indicates that a 'regcall'-type function specifies that an argument is to be passed in one of the following disallowed registers:

- The frame pointer register. This is the register that is used to access a function's local variable.
- The data pointer register. This is the register that is used by small-code or small-data modules to access data.

`171: %%arg not found`

Within a C module's **#asm** code, C function arguments and local variables can be referred to by preceding the C name with %%. For example:

```
void f(long var)
{
#asm
  mov.l       %%var,d0
#endasm
}
```

This error indicates that the name difined with %% is not an argument or local variable of the C function.

`172: invalid return type for regcall-type function`

This error indicates that the prototype for a 'regcall'-type function defines the function to return one of the following disallowed types:

- a structure;
- A floating point argument that won't fit in a 32-bit register.

Fatal Compiler Error Messages

If the compiler encounters a "fatal" error, one which makes further operation impossible, it will send a message to the screen and end the compilation immediately.

`Can't open file '%%' for input`

The input file specified in the command line does not exist on the disk or cannot be opened. A path or drive specification can be included with a filename according to the operating system in use.

`Cannot create output file '%%'!`

The compiler was unable to create an output file. On some systems, this error could occur if a disk's directory is full. It may also occur if the disk is locked.

`Dump file not found!`

This error indicates a failure on attempting to open the specified dump file used with the **-h** option. Check to make sure the correct name was used on the command line following the **-h** option.

`Error creating dump`

This indicates that an error occurred in trying to open the dump file specified following the **-ho** option. Check to make sure the disk is not locked or full.

`Error reading dump file!`

This error indicates that a precompiled dump file that was opened for reading specified a length field that was longer than that actual dump file. The dump file has been corrupted and should be rebuilt.

`Illegal '-%%' option:`

The compiler has been invoked with an option letter which it does not recognize. The manual explicitly states which options the compiler will accept. The compiler will specify the invalid option letter.

`Illegal '-h' option: '%%'`

The **-h** compiler option must be followed by the letter 'o' or 'i', to specify that the compiler is to output or input a precompiled header file, respectively. This error indicates that some other letter followed the **-h**.

`Illegal 3.6 '+x' option: '%%'`

All the **+x** options in the v3.6 compiler have been replaced by new options. Thus, this error message is displayed whenever a module is compiled with a **+x** option.

`Illegal 3.6 option: '%%'`

The version 5.0 compiler will accept 3.6a options if the **-3** option was specified. Consult a 3.6 manual for specific option details. also see discussion of **-3** and **-5** options.

`Illegal "%%" option: "%%"`

A secondary option was specified that is not recognized in conjunction with the first letter given. The error message gives the primary and illegal secondary option used. See the **Compiler** chapter of the manual, which explicitly states the options that the compiler will accept.

`Invalid register in -YF option`

The **-yf** option is used to define the address register that's to be used as the frame pointer. For example, **-yfa4** specifies that **a4** is to be used. This error message indicates that an invalid register was specified.

`Invalid register in -YD option`

The **-yd** option is used to define the address register that's to be used as the data pointer. For example, **-yda4** specifies that **a4** is to be used. This error message indicates that an invalid register was specified.

`More than one output filename was specified`

Output from the compiler can only be directed to one file. More than one **-o** option was found.

`Multiple '-h' options specified`

Only one **-hi** or **-ho** option can be specified. Indicates that the **-h** option has already been used with a dump file name. The compiler only accepts one dump file as input and cannot generate an output dump file if one has already been specified for input. Use the **#include** directive multiple times in a single file to include all the source files that are to be precompiled, and then precompile that single file.

`Multiple input files specified`

More than one input file was specified.

`Need valid register here`

The register specified in the **equr** directive must be a valid **mc68000** register name.

`No input file was specified!`

While the compiler was able to open the input file given in the command line, that file was found to be empty.

`Out of disk space!`

An error occurred in closing the assembly output file being written by the compiler. The compile therefore failed and the assembly file was deleted. Make additional room on the current disk or the one to which the CCTEMP environment variable is pointing.

`Out of memory!`

Since the compiler must maintain various tables in memory as well as manipulate source code, it may run out of memory during operation. The only solutions are to add more memory to your computer or to divide the source file into modules which can be compiled separately. These modules can then be linked together to produce a single executable file.

`Outbyte of 0x%%!`

A non-ASCII character with a value greater than 0x7f was encountered in the source file. An example would be characters obtained with the option key depressed.

`Pre-compiled header not in proper format!`

This error indicates that a precompiled dump file opened for reading did not contain the correct information to indicate that it is a valid dump file. The file is either not a dump file or the dump file has become corrupted and should be rebuilt.

`Pre-compiled header uses the wrong size int`

The compiler will treat ints as 16 bits or 32 bits according to option settings. When a pre-compiled header file is used, it is mandatory that equivalent data size options be in effect.

`Premature end of file in macro definition`

A macro definition must be terminated with the **endm** directive.

`Redefining label type`

An **equ** or **set** cannot be used to assign a value to a label if that label has already been used in some other capacity.

`Symbol required`

Following the **bss, global,** or **public** directives a symbol is required. In the case of **bss** and **global** it labels the reserved storage area. For **public,** it indicates the symbol is visible to other modules.

`Too few arguments for 'c' option`

The listed option requires one or more additional arguments. For example, the compiler expected to find the output file name following the **-o,** but did not find it. The output file name must follow the option letter, and the name of the file to be compiled must occur last in the command line.

`Too many -i options`

The number of include file paths has exceeded the maximum of 16 allowed. Try rearranging the include files into fewer directories to avoid the need to specify so many include file paths.

`Unable to execute as`

An attempt to launch the assembler failed. Check to make sure the assembler has not been renamed and that it can be located in your current execution path.

Internal Compiler Error Messages

The following fatal compiler error messages are internal. None of these errors should normally occur. If you get one, it may point to an internal compiler problem. Please contact Technical Support to report the problem; include some sample code to demonstrate it.

```
Attempt to use expression tree entry twice
Attempt to release item already free
Bad arginfo
Bad cast - 'cast'
Bad op in cgen: 'op'
Bad operator in donode!
Bad value in outarg: opr
Bf_shift didn't work
Optim failure.
Out of registers!
```

Assembler Error Messages

Assembler error messages are broken down into the following four categories:

Opcode Error Messages
Directive Error Messages
Fatal Assembler Error Messages
Syntax Error Messages

Error messages are listed in alphabetical order under each of these sections. Detailed explanations of each error message follow after the summary lists.

Opcode Error Messages

```
68020 addressing mode not valid
Additional argument expected
An, Dn, modes not supported
Bad argument
Bad size for expression
Closing register must be greater
Data register missing from register pair
Field width must be in range (0-31)
Field width must be in range (1-32)
Fourth argument must be An
FP register missing from register pair
Illegal addressing format
Illegal argument expression
Illegal extension on opcode
Illegal function code (mc68851)
Illegal immediate used with 68881 opcode
Invalid argument
Missing ':' in bit field syntax
Missing '}' in bit field syntax
Missing argument
Must be 16 bit
Must be 8 bit
Need address register here
Need closing register
Need data or address register as index
Need data register here
Need FP control register here
Need FP register list as one of the operands (mc68881)
```

Need opcode, directive or macro name here
Need register
Need register here
Need register list as one of the operands
Offset can't be data reference
Only .S and .D immediate operands currently supported
Only W and L are valid modifiers
Opcode illegal for this processor
Opcode operands did not match
Opcode extension size did not match
Pc relative out of byte range, %ld
Pc relative out of word range, %ld
Pmove PSR,(ea) not allowed (mc68851)
Post incrementing illegal in this mode
Scale must be 1, 2, 4, or 8
Second argument must be immediate
Size extension not allowed on this opcode
Syntax error on last arg of CAS2
Third argument must be immediate
Unimplemented opcode
Wrong register type in list

Directive Error Messages

Conditional else directive out of place
Conditional end directive out of place
Equate directive requires a label
Expression must be absolute
Illegal character in directive arguments
Illegal character in conditional
Invalid expression for defined storage
Invalid expression for equate directive
Invalid expression for set directive
Invalid operator in evaluate - %d
Label mismatch
Macro end out of place
Macro name collision with previously defined name
Multiply defined symbol
Name already defined
Need 'code' or 'data' here
Need absolute value here

```
Need file name here
Need name for macro
Need register list here
Need second argument as well
Need symbol for definition check
Need valid register here
Non-subtraction expression with multiple symbols
Premature end of line in conditional string
Premature end of file in macro definition
Redefining label type
REG directive must contain register list
Register list directive requires a label
Requires two strings
Right side of test must match left side
Set directive requires a label
Symbol required
Syntax error on symbolic Debugger line
Unable to open include file
Unimplemented assembler directive
```

Fatal Assembler Error Messages

```
Can't nest macro definition
Can't open input file
Can't open output file
Ending PC's differ
Error writing to listing file
Error writing to object file
Includes nested too deep
Internal error in squeeze algorithm - %d
No input file specified
Out of memory
Too many -i options
Unable to create listing file
Unable to create output file
```

Syntax Error Messages

For consistency, the syntax errors listed below are also listed again in the explanation section, but since these messages are self explanatory, no further explanation is provided.

Please check your typing.

```
Bad '('
Bad character in line
Bad syntax
Extra characters on line!
Illegal '['
Illegal character in string
Missing ')'; Where's the ')'
Missing trailing parenthesis;
Missing ']'; illegal ']';
Unknown opcode or directive
Unknown token in expression
```

Explanation of Assembler Error Messages

This section further explains the error messages generated by the Assembler.

Opcode Error Messages

`68020 addressing mode not valid`

To enable assembly of 68020 addressing modes insert the directive:

machine mc68020

`Additional argument requested`

The instruction requires two or more arguments, each being separated by a comma.

`An, Dn, modes not supported`

Certain 68851 opcodes will not accept **a***n* or **d***n*. (See your 68851 reference manual).

`Bad argument`

Legal directive argument is expected.

`Bad size for expression`

Occurs when you try to use an illegal size extension.

`Closing register must be greater`

The order in a register list must be from **d0** to **d7** followed by **a0** to **a7**.

`Data register missing from register pair`

Data register required after colon (:) in 64 bit multiply or divide, or to specify the width in bit field instructions.

```
Field width must be in range (0-31)
Field width must be in range (1-32)
```

Bit fields are used in the following 68020 instructions:

bfchg, bfclr, bfexts, bfextu, bfffo, bfins, bfset, bftst

These instructions operate on a string of consecutive bits in a bit array.

The syntax for a bit field is:

{*offset:width*}

The braces and colon must be included as shown. *offset* and *width* parameters must either be data registers or absolute expressions. *offset* must be in range (0-31). *width* must be in range (1-32).

`Fourth argument must be An`

If an optional fourth argument is specified for the instructions **ptestr** or **ptestw**, the argument must be an address register to hold the address of the last descriptor successfully fetched.

`FP register missing from register pair`

Second **fp** register required after colon for 68881 instructions like **`fsincos.x`** which returns two results.

`Illegal addressing format`
`Illegal argument expression`

The address mode you are trying to use is not supported by the opcode.

`Illegal extension on opcode`

The instruction being used does not support the extension specified. Check your 680x0 reference manual to determine which extensions are allowed. The extensions include (**l, s, x, p, w, d, b**), but not all extensions are available for each instruction.

`Illegal function code (mc68851)`

Function codes must be expressed in any of the following registers:

d*n*, **sfc**, **dfc**, or as an immediate four bit number.

Examples:

```
ploadr d4,(a5)
ploadr sfc,(a5)
```

`Illegal immediate used with 68881 opcode`

Use extensions **.b**, **.w**, and **.l** for immediate values.

`Invalid argument`

Legal directive argument is expected.

`Missing ':' in bit field syntax`
`Missing '}' in bit field syntax`

Bit fields are used in the following 68020 instructions:

bfchg, bfclr, bfexts, bfextu, bfffo, bfins, bfset, bftst

These instructions operate on a string of consecutive bits in a bit array.

The syntax for a bit field is:

{offset:width}

The braces and colon must be included as shown. *offset* and *width* parameters must either be data registers or absolute expressions. *offset* must be in range (0-31). *width* must be in range (1-32).

`Missing argument`

Directive argument is expected.

`Must be 16 bit`

Data must be 16 bit (**$0000 - $ffff**)

`Must be 8 bit`

Data must be 8 bit (**$00 - $ff**)

`Need address register here`

Address register (**a0, a1, ... a7**) required in the operand field. This is often the case for indirect addressing modes.

`Need closing register`

A dash (-) in a register list must be followed with a ending register.

`Need data or address register as index`

The format of the index operand for 68020 register indirect with index modes is *xn.size*scale*. (See your 68020 reference manual.)

`Need data register here`

If error pertains to bit fields refer to error message:

```
field width must be in range (0-31)
```

If not, many opcodes need a data register (**d0,d1,d2...d7**) in the operand field. You can usually substitute **adda** for **add**, **suba** for **sub** and **cmpa** for **cmp**. (Consult your 680X0 reference manual.)

`Need FP control register here`

fp control registers **fpcr**, **fpsr**, and **fpiar** cannot be combined with other registers in a register list.

`Need FP register list as one of the operands (mc68881)`

fp register lists are used with **fmovem**. The syntax for a **fp** register list is any combination of **fpcr**, **fpsr**, and **fpiar**, with individual register names separated by a slash "/".

Examples:

```
fpcr/fpsr
fpcr/fpiar/fpsr
```

`Need opcode, directive or macro name here`

Label names must be defined starting in the first column of the line.

`Need register`

A register list requires that one or more registers be specified.

`Need register here`

Address or data register required in the operand field. Check your 680x0 reference manual to determine which registers are allowed.

`Need register list as one of the operands`

A register list is used with **movem.**

Examples:

Register List	Meaning
d3-d7	**d3, d4, d5, d6,** and **d7**
d5/a5	**d5** and **a5**
d2-d4/a0-a3	**d2, d3, d4, a0, a1, a2,** and **a3**

`Offset can't be data reference`

Must use an address register (**a0, a1, ... a7**) for indirect addressing.

`Only .S and .D immediate operands currently supported`

For an extended .X operand, use a hex value starting with a **$**.

`Only W and L are valid modifiers`

The format of the index operand for 68020 register indirect with index modes is *xn.size*scale*. (See your 68020 reference manual.)

`Opcode illegal for this processor`

The instruction is not valid for the MC68000 or the processor specified. Use the **machine mc68010, machine mc68020, mc68881,** or **mc68851** directives.

`Opcode operands did not match`

The address mode you are trying to use is not supported by the opcode.

`Opcode extension size did not match`

Occurs when you try to use an illegal size extension.

`Pc relative out of byte range, %ld`

A byte relative instruction is attempting to reference a point greater than 128 bytes away from the current location.

`Pc relative out of word range`

A word relative instruction is attempting to reference a point greater than 32K bytes away from the current location.

`pmove PSR,(ea) not allowed (mc68851)`

(consult your 68851 reference manual)

`Post incrementing illegal in this mode`

Post incrementing (a*n*)+ is only legal with certain opcodes.

Example:

```
add a1+,d0                    ;ERROR
add (a1)+,d0                  ;OK
```

`Scale must be 1, 2, 4, or 8`

The format of the index operand for 68020 register indirect with index modes is *xn.size*scale*. (See your 68020 reference manual.)

`second argument must be immediate`

Should only occur when using **pflush** (**mc68851**). The mask must be immediate.

`Size extension not allowed on this opcode`

Many 680x0 opcodes have only one size or are specified 'unsized'.

Example:

```
pea.w (a0)                    ;ERROR
pea (a0)                      ;OK
```

`Syntax error on last arg of CAS2`

Check the syntax for an indirect multiple argument destination.

`Third argument must be immediate`

The **mc68851** instructions **ptestr** and **ptestw** require a third argument indicating the depth to which the translation table is to be searched. This must be an immediate.

`Unimplemented opcode`

Assembler does not understand opcode. Check your syntax.

(Note: The mnemonic **trap***cc* has been changed to **t***cc* in the parameterless form, and **tp***cc* is used when immediate data is specified. Similar changes have been made for **ftrap***cc* (mc68881) and **ptrap***cc* (mc68851).)

`Wrong register type in list`

Some instructions only allow the use of certain registers. Check your 680x0 reference manual to determine which registers are allowed. For example, control registers generally are not allowed in a register list.

Directive Error Messages

`Conditional else directive out of place`

else directive must have a matching **if** directive.

`Conditional end directive out of place`

endc must be used in conjunction with an **if** directive.

`Equate directive requires a label`

A label must precede all **equate** directives.

Example:

```
$4000    equ   tempvar      ;ERROR
tempvar  equ   $4000        ;OK
```

`Expression must be absolute`

Certain size and value parameters for directives must be absolute and not relative.

`Illegal character in directive arguments`

Directives must be carefully described. (See the **Assembler** chapter.)

`Illegal character in conditional`

These errors pertain to conditional directives.

`Invalid expression for defined storage`

The value of the size field in the **ds** directive must be an absolute expression.

`Invalid expression for equate directive`

An absolute value is a constant expression.
An absolute value must follow a **set** directive.

Examples:

```
rex      equ   1                ;OK
spot     equ   2                ;OK
fido     equ   rex+spot         ;OK
rover    equ   rex+(a4)         ERROR
```

`Invalid expression for set directive`

(see: Invalid expression for **equate** directive)

`Invalid operator in evaluate - %d`

Valid operators in expressions include:

+, -, *, /, >, <, &, |, !, ^, ~, //

`Label mismatch`

The position of the label in pass2 does not match its location calculated in pass1.
Try assembling the module using a **-n** to turn off optimizations.

`Macro end out of place`

The **endm** directive must be used in conjunction with the **macro** directive. All directives and op-codes must be preceded by at least one space or tab.

`Macro name collision with previously defined name`

Each macro definition must use a unique name to identfy it.

`Multiply defined symbol`

Every label must have a unique name.

`Name already defined`

Every label must have a unique name.

`Need 'code' or 'data' here`

The **near** and **far** directives must specify 'code' or 'data'

Examples:

```
near    code
far     data
```

`Need absolute value here`

An absolute value is a constant expression.
An absolute value must follow a **set** directive.

Examples:

```
rex     equ   1              ;OK
spot    equ   2              ;OK
fido    equ   rex+spot       ;OK
rover   equ   rex+(a4)       ERROR
```

`Need file name here`

include directives must be specified as:

include *<filename>*
include *"filename"*

`Need name for macro`

A **macro** directive must be followed by a symbol which is used to name the macro.

`Need register list here`

Register list directives must be specified as follows:

label **reg** *register_list*

Macros must be defined as:

[*label*] **macro** *symbol*

The macro name is the label preceding the **macro** directive or the symbol following it. Every macro name must be unique.

`Need second argument as well`

The **cnop** directive requires two arguments.

`Need symbol for definition check`

These errors pertain to conditional directives.

`Non-subtraction expression with multiple symbols!`

The only valid computation using two symbols is the calculation of their difference.

`Premature end of file in macro definition`

A macro definition must be terminated with a **endm** directive.

`Premature end of line in conditional string`

These errors pertain to conditional directives.

`Redefining label type`

The label in question has already been defined as being used for some other type of operation.

`REG directive must contain register list`

These errors pertain to conditional directives.

`Register list directive requires a label`

These errors pertain to conditional directives.

`Requires two strings`

These errors pertain to conditional directives.

`Right side of test must match left side`

These errors pertain to conditional directives.

`Set directive requires a label`

See:

`Equate directive requires a label`

`Syntax error on symbolic Debugger line`

The compiler generated information for **sdb** isn't rcognized by the assembler. Make sure the version of the assembler being used matches that of the compiler.

`Unable to open include file`

Include directives must be specified as:

include *<filename>*
include *"filename"*

`Unimplemented assembler directive`

The assembler does not understand. Check your syntax.

Fatal Assembler Error Messages

`Can't nest macro definitions`

The current macro definition must be terminated with an **endm** before beginning a second one.

`Can't open input file`

The source file could not be opened. Check to make sure the spelling is correct and that the file actually exists.

`Can't open output file`

The output file specified with the **-o** option or the default could not be created by the operating system. Possible reasons include a locked or full disk.

`Ending Pc's differ`

The size of the module in pass2 doesn't match the value calculated in pass1. Try assembling the module using a **-n** to turn off optimizations.

`Error writing to listing file`

Some kind of an operating system error occured while attempting to write out the listing file. One possible cause would be if disk is full.

`Error writing to object file`

Some kind of an operating system error occured while attempting to write out the object file. One possible cause would be if the disk is full.

`Includes nested too deep`

Include files may not be nested deeper than 7 levels.

`Internal error in squeeze algorithm - %d`

Internal assembler error. Call tech support with a small sample of the code that generated this error. A temporary work around for this error is to use the **-n** option which turns off optimizations.

`No input file specified`

The assembler was invoked without giving a assembly source file to be assembled.

`Out of memory!`

The assembler ran out of memory while assembling a file. To resolve the problem either split the source file into two or more files to reduce the number of symbols for which space is required or use a computer that has more available memory.

`Too many -i options`

The total number of directory paths that may be searched cannot exceed 16.

`Unable to create listing file`

The listing file generated with the **-l** option could not be created by the operating system. Possible reasons include a locked or full disk.

`Unable to create output file`

The output file specified with the **-o** option or the default could not be created by the operating systems. Possible reasons include a locked or full disk.

Syntax Error Messages

Please check your typing.

```
bad '('
Bad character in line
bad syntax
extra characters on line!
illegal '['
Illegal character in string
Missing ')'; Where's the ')'
Missing ']'; illegal ']';
Missing trailing parenthesis;
Unknown opcode or directive
Unknown token in expression
```

Linker Error Messages

This section lists the error messages that the linker may display as it creates an executable program. The messages are grouped according to the source of the errors that cause them. Elements that are variable are enclosed by angled brackets. A more detailed explanation of each message follows after the summary list.

When started, the linker first displays a message on the screen that indicates that the linker is loaded and running. If everything goes well, the linker prints several messages on the screen listing the sizes of the program segments; then the linker finishes. The linker may encounter an error while it is running, in which case it sends a message to the screen.

Errors are reported at a variety of points during the linking process. ln generates an executable program in two stages, known as pass1 and pass2. The size messages are printed at the end of pass1, so any errors occurring after that are detected during pass2 of the linker.

Following is a list of the messages that the linker generates in response to an error.

Note: Only one file type option is permitted.

Command Line Error Messages:

```
Cannot have nested -f options.
Invalid overlay number
No input given!
Too few arguments in -f file: filename
Too few arguments in command line
Unknown option 'c'
```

I/O Error Messages

```
Bad symbol typing information
Can't create .dbg symbol file filename
Can't open filename, err = errno
Cannot create output file: filename
Cannot create symbol table output
Cannot open -f file: filename
Cannot write output file
Couldn't open %s in pass2!
Couldn't read object file %s!
Corrupted object files
Error creating symbol listing file
Error reading module on pass2!
Error while lseeking output file!
```

Error reading/writing output file!
Invalid operator in evaluate *hex value*
Library format is invalid!
Line displacement too large at line = %d
Not an object file
Object file is bad!
Symbol name too long

Memory Use Error Messages

Out of memory
Too many symbols!

Source Code Error Messages

Absolute reference from segment to segment
Attempt to perform relocation in overlay code
Attempt to use SMALL reference to $%lx
Attempted to write outside of file bounds
Bad struct index!
Branch out of range @pc=*addr*
Can't do PC rel as dest @pc=%lx
Can't do relocation from data to nonroot code
Can't find STKSIZ
Can't find _Storg_!
Data ref to overlay code not in jump table
Data reference out of range
pass1(*hex value*) and pass2(*hex value*)values differ:
Premature EOF in object module at offset 0x%lx!
Program is too large to link
Short branch to next location @pc=*addr*
symbol name multiply defined
symbol type differs on pass two: *symbol name*
Total data size >64K! Too large for small data model!
Undefined symbol: *symbol name*
Unknown loader item (%02x)!

Internal Linker Error Messages

Error setting relocatable references

Explanation of Linker Error Messages

Command Line Errors

`Cannot have nested -f options.`

At the command line level any number of command files can be specified by using the -f option. However, none of these files can contain the -f option. A command file cannot invoke another command file

`Invalid overlay number`

Overlays must be positive integers.

`No input given!`

The linker quits immediately if it is not given any input to process.

`Too few arguments in -f file: filename`

An option letter specified in the file, *filename*, requires a value or name to follow it. If an option appears at the end of the file, its associated value may not appear back on the command line.

`Too few arguments in command line`

Several of the linker options have an associated value or name, such as -b 2000. If a needed value is missing, the linker displays this message and dies.

`Unknown option 'c'`

You have used an option letter(*c*) that the linker does not recognize. The linker ignores only the letter; it preserves everything else on the command line and the linker tries to execute what it can interpret. See the **Linker** chapter for a list of options that are supported.

I/O Errors

`Cannot create output file: filename`

This message usually indicates that all available directory space on the disk has been exhausted.

`Cannot create symbol table output`

Option -t is given in the command line, but the file containing the linkage symbol table cannot be written to disk. It is possible that there is no more space on the disk.

`Cannot open -f file: filename`

A file given with option -f cannot be opened.

`Cannot write output file`

This message usually indicates that all available directory space on the disk has been exhausted.

`Can't create .dbg symbol file filename`

Option -t is given in the command line, but the file containing the linkage symbol table cannot be written to disk. It is possible that there is no more space on the disk.

`Can't open filename, err = errno`

If any file in the command line cannot be opened, this message is sent to the screen, specifying the *filename* and the current value of *errno*.

`Couldn't open %s in pass2!`

Object module file that was read during pass1 couldn't be opened for pass2. On a multiuser system the file may have been deleted by another process.

`Couldn't read object file %s!`

Attempt to read an object file failed. This could be due to a corrupt file or a hardware error.

`Corrupted object files`

This is the most explicit indication that an object file in the linkage is corrupted. Recompile and assemble the source file. A bad object file will not be discovered until the second pass of the linker.

`Error creating symbol listing file`

Option -t is given in the command line, but the file containing the linkage symbol table cannot be written to disk. It is possible that there is no more space on the disk.

`Error reading module on pass2!`

Message indicates that a module is corrupted between pass1 and pass2. On a multiuser system, it is possible that another user changed the file while the linker was running. Otherwise, the error is probably due to hardware failure.

`Error while lseeking output file!`

Invalid attempt to position beyond the end of th program file being generated.

`Error` *`errno`* `reading/writing output file`

An error reading or writing the output file probably means there is no more disk space available. In particular, a block of the output file was written to disk and then could not be read back. The current value of *errno* is given in these messages.

`Invalid operator in evaluate` *`hex value`*

Unless you changed the object code by hand, the file is corrupted.

`Library format is invalid!`

A library in the linkage has been corrupted.

`Line displacement too large at line = %d`

Source-level debugging information is in error. Make sure you have a current version of the linker and the compiler.

`Not an object file`

A file given to the linker does not contain relocatable object code that ln can process. For instance, a source file may have been included in the link.

`Object file is bad!`

This is the most explicit indication that an object file in the linkage is corrupted. Recompile and assemble the source file. A bad object file will not be discovered until the second pass of the linker.

`Symbol name too long`

Source-level debugging information is in error. Make sure you have a current version of the linker and the compiler.

Errors in Use of Memory

`Out of memory!`

The linkage process needs memory space for ln and global and local symbol tables, and approximately 5K for buffers. Just as with compilation, most memory use is devoted to the program software and symbol tables. Since ln is not especially large, only an extremely complicated linkage might run out of memory.

`Too many symbols!`

This is another way of saying that not enough memory is available for the symbol tables needed for the linkage.

Errors Arising From Source Code

`Absolute reference from segment to segment`

Segments may only reference other segments via the jump table. Check your assembler code, the compiler will never generate these.

`Attempt to perform relocation in overlay code`

The only segments of a program that can contain addresses that must be relocated when the program is loaded are the program root code segment and its initialized data segment. The relocation of these two segments is performed when the program is first loaded, by the Manx-supplied startup code.

`Attempt to store out of bounds`

This error should not occur. It indicates a linker bug.

`Attempt to use SMALL reference to $%lx`

A segment relative reference greater than 32K was attempted.

`Bad struct index!`

Exceeded the 512 maximum number of structure templates allowed in a module.

`Branch out of range @pc=addr`

A **branch** or **jump** instruction has a target address that is beyond its range. This error should not be generated from C programs.

`Can't do Pc rel as dest @pc=%lx`

Attempt to use a PC relative reference as the destination for a 68000 memory altering instruction. No such addressing mode is available for these instructions.

`Can't do relocation from data to nonroot code`

The only permissible data references are via the jump table. Therefore data can only be assigned the address of a function's entry point.

`Can't find STKSIZ`
`Can't find _Storg_!`

These errors should not occur. If they do, please report them.

`Data ref to overlay code not in jump table`

This error is caused by C programs that attempt to initialize a global pointer to a static function, where the function is contained in an overlay. Such initialization is permitted when the function is located in the root segment, but not when it is in another segment.

`Data reference out of range—remake with large data model`

Initialized data plus uninitialized data exceeds 32K. Recompile all C source modules with **-md**, reassemble assembly modules with **-d**, and relink with **cld.lib** (large data). An alternative approach would be to dynamically allocate some of your data at run time.

`Pass1(hex value) and pass2 (hex value)values differ:`

Either of these errors may be generated during pass2 when error 24 appeared in pass1. They may be considered a confirmation of what was discovered in pass1 of the linker.

`Premature EOF in object module at offset 0x%lx!`

Caused by a corrupt or truncated object. It may also occur if the size of the object file during pass2 differs from the value calculated during pass1.

`Program is too large to link`

This error should not occur. It indicates a linker bug.

`Short branch to next location @pc=addr`

The 68k processor does not accept instructions of this type. This error should not occur, since the Manx assembler detects such instructions and removes them from the object code.

`symbol_name` `multiply defined`

A global symbol is defined more than once. For instance, if two functions are accidentally given the same name, this message is generated.

`Symbol type differs on pass two:` *`symbol name`*

Either of these errors may be generated during pass 2 when error 24 appeared in pass 1. They may be considered a confirmation of what was discovered in pass 1 of the linker.

`Total data size >64k! Too large for small data model!`

When any of a program's modules is compiled to use the small data memory model, that module's data is accessed using an offset from the program's data pointer register. The area accessible in this way is at most 64kb. Thus, this error message indicates that at least one of a program's modules uses the small data memory model and that the program has more than 64kb of data.

`Undefined symbol:` *`symbol_name`*

A global symbol name remained undefined. This is commonly a function that has been referenced in the source code but not included anywhere in the link.

`Unknown loader item (%02x)!`

Object code contains an instruction which the linker does not understand. May be due to using an unsupported feature in the assembler or an object module that is not at the same version as the linker.

Internal Errors

`Error setting relocatable references!`

Internal error locating a relocatable reference. Call tech support with a sample of the object code that caused the problem.

Chapter 13 - Technical Support

We have put together a set of guidelines to help you take the most advantage of the technical support service offered by Manx. We ask that you read and follow these guidelines to enable us to continue to give you quality technical support.

Have Everything With You

Try to be organized. When using our phone support, have everything you need with you at the time you call. Our goal is to give you the help you need without keeping you on the phone too long. This can save you a lot of time, and if we can keep the calls as short as possible, we can take more calls each day. This can be to your advantage on days when we are busy and it is hard to get through. Also, have the following information ready when you call technical support. We will ask you for this information first.

- **Your name**. This is necessary in case we need to get back to you with additional information.

- **Phone number**. In case we have additional information we will be able to contact you. This will never be given to anyone, so you need not worry.

- **Name of the product you are using, *and its SERIAL NUMBER*.** If you have a cross compiler, please tell us both host and target, even if the problem is with only one side of the system.

- **The revision number of the product you are using.** This should include a letter after the number: i.e., 3.20d or 1.06d. **THIS IS VERY IMPORTANT.** The full version number may be found on your distribution disks or when you run the Compiler.

- **The operating system you are using, and also the version.**
- **The type of machine you are using.**
- **Anything interesting about your machine configuration.** i.e., ram disk, hard disk, disk cache software, etc.

Know What Question You Wish To Ask

If you call with a usage question, please try to have your questions narrowed down as much as possible. It is easier and quicker for all to answer a specific question rather than a general one.

Isolate The Code That Caused The Problem

If you think you have found a bug in our software, try to create a small program that reproduces the problem. If this program is small enough, we will take it over the phone; otherwise we would prefer that you mail it to us, using the supplied problem report, or leave it on one of our bbs systems. Once we receive a "bug report," we will attempt to reproduce the problem and, if successful, we will try to have it fixed in the next release. If we cannot reproduce the problem, we will contact you for more information.

Use Your C Language Book And Technical Manuals First

Manx Technical Support is happy to help you with problems or questions relative to the operation of our products. If you are having difficulty with the C language syntax or a C programming problem in general, please check with a C language programming book, or contact a local university or user group. If you have questions about machine specific code, i.e., interrupts or DOS calls, check with the technical reference manual for that machine and/or its operating system manual.

When To Expect An Answer

A normal turn around time for a question is anywhere from two minutes to 24 hours, depending on the nature of the question. A few questions, like tracing compiler bugs, may take a little longer. If you can call us back the next day, or when the person you speak with in technical support recommends, we will have an in-depth answer for you. But normally we can answer your questions immediately.

Use Our Mail-in Service

It is always easier for us to answer your question if you mail us a letter. (We have included copies of our problem report form for your use.) This is especially true if you have found a bug with our

compiler or other software in our package. If you do mail your question in, try to include all of the above information, and/or a disk with the problem. Again, please write small test programs to reproduce possible bugs. See below for the Manx mailing address.

Updates, Availability, Prices

If you have questions about updates, availability of software, or prices, please contact the appropriate department. See below for a complete listing of the Manx addresses and telephone numbers.

Bulletin Board System

For users of Aztec C, we have a bulletin board system available. See below for the different bulletin board numbers.

Follow the questions that will be asked after you are connected. When this is done, you will be on the system with limited access. To gain a higher access level, send mail to SYSOP. Include in this information your serial number and what product you have. Within approximately 24 hours you should have a higher access level, provided the serial number is valid. This will allow you to look at the various information files and upload/download files.

To use the bulletin board best, please do not put large (>8 lines) source files onto the news system, which we use for an open forum question/answer area. Instead, upload the files to the appropriate area, and post a news item explaining the problem you are having. Also, the smaller the test program, the quicker and easier it is for us to look into the problem, not to mention the savings of phone time. When you do post a news item, please date it and sign it. This will be very helpful in keeping track of questions. Try to do the same with uploaded source files.

Phone Support

And finally, telephone technical support is provided with the purchase of your Aztec C for 90 days from date of purchase. It is available between 10:30-12 noon and 3-5 p.m. eastern standard time. (Times subject to change without notice; for the correct telephone number, see below.) Phone support is available to registered users of Aztec C.

These guidelines will aid us in helping you quickly through any roadblocks you may find in your development.

Manx Software Systems

Address:

Manx Software Systems
P.O.Box 55
Shrewsbury, NJ 07702

Phones:

Technical Support	(908) 542-1795
Sales (Domestic)	(800) 221-0440
Sales (International) & Updates	(908) 542-2121
FAX:	(908) 542-8386
Bulletin Board: 300/1200 bps (all products)	(908) 542-2793

Note: The above information is subject to change without notice.

On January 1, 1991: Our area code was changed to **908**.

Manx Problem Report

Date:____/____/____ Name:_____

Phone: 1-(____) _____-_____

Company:_____

Address:_____

Product: Aztec C86: ❑For DOS Applications

❑For Embedded Systems Applications

Version #:_____ Serial #:_____

Target's Operating System:_____

Machine Configuration: _____

Description of problem

Include description of problem and efforts to fix it.

Index

!

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z