

Development Software for the Amiga

TM



M

A

N

XTM

Aztec C User Guide
Aztec C Reference Manual
Aztec C Library Manual
Aztec C UniTools
Aztec C Source Level Debugger

Before you do anything else...

READ THIS PAGE!

This is your Aztec C documentation. Upon close inspection, you may find something a little unusual about this book. That is because this book is actually *five separate manuals* which have been bound together as *one* book for your convenience and accessibility.

The five books included here are:

- Book 1: Aztec C User Guide**
- Book 2: Aztec C Reference Manual**
- Book 3: Aztec C Library Manual**
- Book 4: Aztec C UniTools**
- Book 5: Aztec C Source Level Debugger**

On the next page is a list of the chapters that are found in these five manuals. Since these manuals are essentially five separate units, each one comes with its own Table of Contents and its own Index. The chapters also have "thumb tabs" for easier access. If you turn this book on its side, you will see these thumb tabs, as well as the solid black lines that separate the five main sections.

Your Aztec C documentation is designed to help you through the C programming process, from installing your disks to performing the most complex of tasks. To use this documentation most effectively, you should read **Chapter One**, the *Overview*, of *The Aztec C User Guide* before you proceed.

We hope that you will enjoy using this newly revised documentation. We welcome any comments or suggestions you may have that will help us as we prepare future releases. Send any remarks about the documentation to the attention of the Documentation Department.

Thank you, and enjoy! You have chosen the most powerful C development system available for the Amiga, and we are confident that Aztec C will surpass your highest expectations.

Attention: Updates!

If you have updated to this version of Aztec C for the Amiga, refer to the section "Notes to Previous Users" in the *Overview* chapter of the *Aztec C User Guide*. This section includes important information for previous users!

Aztec C for the Amiga

Book 1: Aztec C User Guide

- Chapter 1 - Overview
- Chapter 2 - Getting Started
- Chapter 3 - Aztec Shell Tutorial
- Chapter 4 - Technical Support

Book 2: Aztec C Reference Manual

- Chapter 1 - Overview
- Chapter 2 - Compiler
- Chapter 3 - Language Specifications
- Chapter 4 - Assembler
- Chapter 5 - Linker
- Chapter 6 - Utilities
- Chapter 7 - Debugger
- Chapter 8 - Technical Information
- Chapter 9 - Error Messages

Book 3: Aztec C Library Manual

- Chapter 1 - Introduction
- Chapter 2 - Library Overview
- Chapter 3 - Library Functions
- Chapter 4 - Workbench

Book 4: Aztec C UniTools

- Chapter 1 - Overview
- Chapter 2 - Diff
- Chapter 3 - Grep
- Chapter 4 - Make
- Chapter 5 - Z Editor

Book 5: Aztec C Source Level Debugger

- Chapter 1 - Overview
- Chapter 2 - Getting Started
- Chapter 3 - Tutorial
- Chapter 4 - Commands

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 201-542-2121.) Or, you may write to us at: Manx Software Systems, P.O. Box 55, Shrewsbury, NJ 07702.

- *The C Programming Language* 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* by Samuel P. Harbison and Guy L. Steele, Jr. (Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632)
- *Inside the Amiga* by John Thomas Berry. (Howard W. Sams & Company, 4300 West 62nd St., Indianapolis, Indiana 46268)
- *Inside the Amiga with C* by John Thomas Berry. (Howard W. Sams & Company, 4300 West 62nd St., Indianapolis, Indiana 46268)
- *Amiga Programmer's Handbook* by Eugene P. Mortimore. (SYBEX, Inc., 2344 Sixth Street, Berkeley, California 94710)
- *Programming the 68000 Macintosh Assembly Language* by Edwin Rosenzweig and Harland Harrison. (Hayden Book Company, Hasbrouck Heights, New Jersey 07604)
- *Motorola's MC68000 16-bit Microprocessor User's 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)

- *AmigaDOS Technical Reference Manual*. (Commodore-Amiga, Inc.
1200 Wilson Dr, West Chester, Pennsylvania 19380)
- *Amiga Developers Reference Guide* by David Lai. (Pacific Press,
Post Office Box 611075, San Jose, California 95161-1075)
- *The "Kickstart" Guide to the Amiga*. (ARIADNE Software, Ltd., 273
Kensal Road, London W10 5DB England)
- *Amiga ROM Kernel Manual, Volumes 1 and 2*. (Commodore Business
Machines, Inc., Amiga Technical Reference Series, Addison-Wesley
Publishing Company, Inc., Reading, Massachusetts)
- *Amiga Hardware Reference Manual*. (Commodore Business
Machines, Inc., Amiga Technical Reference Series, Addison-Wesley
Publishing Company, Inc., Reading, Massachusetts)
- *Amiga Intuition Reference Manual*. (Commodore Business
Machines, Inc., Amiga Technical Reference Series, Addison-Wesley
Publishing Company, Inc., Reading, Massachusetts)
- *Amiga ROM Kernel Reference Manual: Libraries and Devices*. (Commodore Business
Machines, Inc., Amiga Technical Reference Series, Addison-Wesley
Publishing Company, Inc., Reading, Massachusetts)
- *Amiga ROM Kernel Reference Manual: Exec*. (Commodore Business
Machines, Inc., Amiga Technical Reference Series, Addison-Wesley
Publishing Company, Inc., Reading, Massachusetts)

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

Aztec C User Guide for the Amiga

**Version 5.0
October 1989**

Copyright © 1988, 1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

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

This software package and document are copyrighted ©1988, 1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

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, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh 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

October 1989	Fifth Edition
December 1988	Fourth Edition
May 1988	Third Edition
April 1986	Second Edition
February 1986	First Edition

Table of Contents

Chapter 1 - Overview

Introduction	1 - 1
AZTEC C CONFIGURATIONS AND DOCUMENTATION	1 - 1
PORTABILITY, DEPENDABILITY, AND SPEED	1 - 3
How To Proceed	1 - 4
About This User Guide	1 - 5
Note to Previous Users	1 - 6
Aztec C Reference Manual	1 - 6
Aztec C Library Manual	1 - 6
Aztec C UniTools	1 - 7
Aztec C Source Level Debugger	1 - 7
Summary	1 - 7

Chapter 2 - Getting Started

System Requirements	2 - 1
Backup the Distribution Disks	2 - 2
Read the README File	2 - 2
Installation Procedure	2 - 3
System Overview	2 - 3
Basic elements of Aztec C	2 - 4
Types of Files	2 - 4
executable files	2 - 4
library files	2 - 4
header files include:	2 - 5
Library Files for Different Data Definitions	2 - 5

Environment Variables	2 - 7
Resident Libraries	2 - 7
Amiga Workbench	2 - 7
User Created Resident Libraries	2 - 8
#pragmas	2 - 8
Pre-compiled Headers	2 - 8
Debuggers - db sdb sdbf	2 - 9
Third Party Software	2 - 9
Freeware, Shareware, and Bulletin Boards	2 - 9
Multiple Segments and Transient Overlays	2 - 10
Mixed C and Assembly Code	2 - 10
QuikFix	2 - 10
QuikFix Compiler Options	2 - 11
QuikFix Z Editor Options and Commands	2 - 11
Batch Processing with QuikFix	2 - 12
Make and QuikFix	2 - 13
Further Information on QuikFix	2 - 13
SYSTEM PERFORMANCE AND COMPATIBILITY	2 - 13
MAKING A WORKING COPY OF AZTEC C	2 - 14
Libraries	2 - 17
Using Aztec C with a Hard Disk	2 - 19
If You Have A Problem	2 - 21

Chapter 3 - Tutorial

Development Environments	3 - 1
WORKING THROUGH THE TUTORIAL	3 - 1
Using the CLI	3 - 2
STARTING AZTEC C	3 - 2
A. Using floppy disks:	3 - 2
B. Using a hard disk:	3 - 2

COMPILE, ASSEMBLING, AND LINKING	3 - 6
ENVIRONMENT VARIABLES	3 - 7
THE RAM DISK	3 - 8
Summary	3 - 9

Chapter 4 - Technical Support

Have Everything With You	4 - 1
Know What Question You Wish To Ask	4 - 2
Isolate The Code That Caused The Problem	4 - 2
Use Your C Language Book And Technical Manuals First	4 - 2
When To Expect An Answer	4 - 3
Use Our Mail-in Service	4 - 3
Updates, Availability, Prices	4 - 3
Bulletin Board System	4 - 3
Phone Support	4 - 4
MANX PROBLEM REPORT	4 - 5
Description of problem	4 - 5

OVERVIEW

GETTING STARTED

AZTEC SHELL
TUTORIAL

TECHNICAL SUPPORT

OVERVIEW

1

Chapter 1 - Overview

Introduction

The Aztec C Software Development System is the ideal tool for both inexperienced and experienced C language programmers. Aztec C is powerful and flexible, offering the freedom to choose from a variety of programs and applications to fit your specific C programming needs.

Inexperienced users can employ the UNIX standard input/output routines to quickly create C programs that will run on the Amiga, without having to use the more complex Amiga Toolbox routines. *Experienced users* can employ extended interfaces with full access to all of the Amiga Workbench and other ROM routines.

Aztec C is fully compatible with the draft proposed ANSI standard and provides extensive support for 68020, 68881, device drivers, Intuition, and other specialized Amiga software and hardware. Numerous examples are provided with your Aztec C package.

AZTEC C CONFIGURATIONS AND DOCUMENTATION

The Aztec C product line for the Amiga offers two core systems and several add-on products, as follows:

- **Aztec C - Professional System**

The Professional level of Aztec C for the Amiga includes the Aztec Shell, Compiler, 68000 Macro Assembler, Overlay Linker, Librarian, Run Time Libraries, Profiler, Full Amiga Workbench Interface, Z editor, and a Portable C Library Interface.

Aztec C offers four different IEEE floating point formats: Manx IEEE double precision emulation, Motorola 68881 coprocessor, Motorola IEEE Floating Point, and Amiga Fast Floating Point. These formats are especially helpful with math programming.

- **Aztec C - Developer System**

The Developer level of Aztec C for the Amiga includes the Aztec Shell, Compiler, 68000 Macro Assembler, Overlay Linker, Source Level Debugger, Librarian, Run Time Libraries, Profiler, Full Amiga Workbench Interface, a Portable C Library Interface, and the complete set of Manx "Unitools."

UniTools is a productivity enhancement package which includes the utilities **make**, **diff**, **grep**, and **z editor**.

make is a program maintenance utility

diff is a source file comparison utility

grep is a pattern-matching program

z is a powerful text editor, similar to the
UNIX vi editor.

Documentation for both the Professional System and the Developer System includes: the *Aztec C User Guide*, the *Aztec C Reference Manual*, the *Aztec C Library Manual*, and *Aztec C UniTools* for the Amiga. Note that if you have purchased the Professional System, you will receive the documentation for UniTools, but only receive the Z editor portion of the software. **make**, **grep**, and **diff** are not included in the Professional System.

The special Aztec C add-on products include:

- **Source Level Debugger**

SDB is Manx's critically acclaimed Source Level Debugger. It is an interactive debugger, designed for fast response and ease in debugging. The windows in SDB display C source and command output separately, with a third window for entering commands. With SDB, you can...

- ...display all active function names
- ...display values of passed parameters
- ...examine variables from any active function
- ...use function or line-by-line tracing
- ...set breakpoints by lines, functions, or variables
- ...see actual C source as it executes
- ...customize the debugging environment with reusable command macros and procedures
- ...and more!

Documentation for SDB is the *Aztec SDB Guide*.

- **Aztec Library Source**

Aztec Library Source includes the original assembly and C source to all routines found in the Aztec Run Time Libraries. Aztec C Library Source does not come with a separate manual; all of the information needed to use Library Source is found in the *Aztec C Library Manual*, which is included with the basic Aztec C package.

PORATABILITY, DEPENDABILITY, AND SPEED

Manx makes several types of Aztec C Software Development Systems. Manx's *native development systems* allow development to be done on the Amiga; Manx's *cross-development systems* allow development to be done on other machines, with the resulting programs downloaded to the Amiga.

Manx also has compilers for developing C programs which run on systems other than the Amiga. Native development or cross development systems

are available for the Macintosh, Atari, Apple II, MS-DOS, and other popular systems.

When you chose Aztec C for the Amiga, you chose dependability. Since 1981, Manx has been acquiring and developing an extensive and comprehensive set of test suites that thoroughly exercise our products to provide the high degree of reliability and dependability that our users expect. Thousands of users have worked with Aztec C for the Amiga since 1986 and have given us their feedback—good and bad. Manx uses these suggestions to continually improve and enhance Aztec C.

Aztec C is also known for its speed. When you use the Aztec compiler, you can specify precompiled header files, thereby speeding up compilation time. If you already have a header file of your own, our compiler allows you to build precompiled header files and use them with our package.

The optimizing assembler is another important factor when you are concerned about the size and speed of your program. It performs branch shortening, as well as other code improvements.

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 very thorough—and a 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 get off to a quicker start.

- Step 1: Read the **readme** file (on disk).
- Step 2: Install your disks, using the installation instructions in Chapter 2, **Getting Started**.

- ❑ Step 3: Familiarize yourself with Aztec C by completing this chapter. Be sure to work through the "hello world" program which Aztec provides step by step in Chapter 3, **Tutorial**.
- ❑ Step 4: Look through both your *Aztec C Reference Manual* and your *Aztec C Library Manual*. Pay particular attention to the **Language Specifications**, **Compiler**, **Assembler**, and **Linker** chapters of the *Reference Manual*. Even if you are an experienced Aztec C user, you should skim through these chapters for the most up-to-date information on Aztec C.
- ❑ Step 5: If you have the Developer System, refer to your *Aztec C UniTools manual* for detailed information on **diff**, **grep**, **make**, and the **z editor**, and your *Aztec C Source Level Debugger Manual* for detailed information on **sdb**.

About This User Guide

This guide gives you the basic information that you need to start working with Aztec C. It consists of this introduction, a guide for getting started, a tutorial, and information on our technical support.

Of course, this manual also includes a detailed **Table of Contents** and a thorough **Index**.

Throughout this manual, we use the following conventions:

input to indicate data entered by the user (e.g., commands, options, and functions)

output to show text that is generated by the computer.

DEFINITION small uppercase bold is used on terms that may be new to the user; most likely will 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.

[*optional*] brackets to show optional information.

Note to Previous Users

If you have used Aztec C for the Amiga prior to the release of Version 5.0, then you should be aware of the many changes and additions that have been implemented. The following paragraphs discuss these changes and direct you to the appropriate chapters for further information.

Aztec C Reference Manual

The most important change to Aztec C for the Amiga is that it is now *fully draft-proposed ANSI standard*. Aztec C's implementation of ANSI C is discussed fully in Chapter 2 of your *Aztec C Reference Manual, Language Specifications*.

It is very important that you read Chapter 3, **Compiler**, particularly with regards to all of the compiler options. This chapter discusses many substantial changes to the compiler.

The following chapters also include changes from previous releases and should be reviewed:

- **Assembler**
- **Utilities**
- **Debugger**
- **Technical Information**
- **Error Messages**

Aztec C Library Manual

The library functions included with your Aztec C system have changed substantially with the draft-proposed ANSI standard. Both the **Library**

Overview chapter and the **Library Functions** chapter reflect these changes and should be reviewed.

You will notice, also, that the **Library Functions** chapter has been improved and expanded, offering a complete synopsis and description of each function. Most functions are listed separately, with frequent references to other related functions.

Aztec C UniTools

With the **make** utility, you will find one minor change: Default rules, when applied to files in other directories, will place the results in those other directories. Previously, the results were placed in the current directory.

Aztec C Source Level Debugger

Refer to Chapter 2, **Getting Started**, and Chapter 4, **Commands**, for changes and new commands.

Summary

You should find that the extended type-checking capabilities of ANSI C make programming easier. And, since ANSI is the standard for the C language, your Aztec C programs will now be more portable than ever before.

GETTING STARTED

2

Chapter 2 - Getting Started

This chapter provides the information you need to begin using Aztec C. The following are the major topics:

- System requirements
- Backing up the distribution disks
- The **readme** file
- Installation Procedure
- System overview
- QuikFix Feature
- System Performance & Compatibility

System Requirements

To efficiently use Aztec C, make sure your hardware and software meet the following specifications:

- Aztec C is compatible with the Amiga 500, Amiga 1000, Amiga 2000, and Amiga 2500. Aztec C is also compatible with all known accelerator boards. Manx recommends at least one floppy disk drive, preferably two. Aztec C is easily installed on a hard disk. Aztec C will run comfortably in 500k of memory.
- SDB, the Source Level Debugger, only uses 90k, but requires a minimum of 256K of available RAM memory. (sdb is included as part of the Aztec C68k Amiga Developer System)

Backup the Distribution Disks

The **diskcopy** program provided with your Amiga will copy all of the distribution disks, or you can use any other utility that copies disks. The disks are not copy protected. It is an excellent idea to leave the original distribution disks write-protected and to make at least one backup copy. If you have a problem with any of the distribution disks, call or FAX Manx Customer Support. The Fax number is 201-542-8386 and the Customer Support number is 201-389-0290.

Walt Disney once said, "We don't make movies to make more money. We make money to make more movies." There are estimates that for every legal copy of commercial software there are three illegal copies. Bootleg software, in general, diminishes the quantity and quality of software available for any machine. Giving away copies is as wrong as selling copies. Support better software by buying it and encouraging others to do so.

Read the README File

You can access the **read.me** file from the CLI environment by inserting distribution disk #1 into drive **df0:** and entering,

```
execute df0:readme
```

The **readme** execute file is one word and there is no ". ". The actual data file has a ":" between "read" and "me".

The **read.me** document provides corrections and additions to information in the manual. Some information will only be found in the **read.me** file. It is important to read this file.

If you want to send the **read.me** document to the printer, use the command:

```
type > PRT: read.me
```

Installation Procedure

The Aztec C system must be installed to a floppy disk or hard disk. To install Aztec C from the CLI environment insert your distribution disk labeled #1 into drive **df0:** and enter

```
execute df0:install
```

The installation procedure will walk you through the steps to create a floppy or hard disk working version of Aztec C.

It may be useful to read the **System Overview** Section below, and the rest of this chapter before performing the installation. A knowledge of the basic system structure and interactions can be useful in executing the install procedure.

The installation procedure will create a default system or a customized higher performance system. If you are new to Aztec C, it is advisable to use the default system. The default system sacrifices some code and execution efficiency, but it chooses options that require the least amount of effort or knowledge. If you require faster, smaller code and faster compile time, then choose the customized version. The system configuration can be changed later by re-installation. The system configuration can also be changed by changing options to the Aztec programs, by changing libraries, or by setting different values for environment variables.

A description of all the files on the distribution disks will be found in the **read.me** file on distribution disk #1.

System Overview

The Aztec C System is highly configurable. Features and characteristics can be enabled, disabled, or changed by user definable options. Some features may also require changing libraries or header files.

These options give you enormous flexibility. They also require careful definition. It is important to understand all of what is required for a

particular feature. Most of the information needed is provided in this manual. Some of the information you need may be in the Amiga system manuals.

The following overview describes features and structures that are useful to know to install, modify, and use the Aztec C System.

Basic elements of Aztec C

Types of Files

executable files	no filename extension
library files	.lib filename extension
header files	.h filename extension

executable files

programs
cc - compiler
as - assembler
ln - linker
sdb - source level debugger
z - source editor
make - project administrator
commands
set - set environment variables
utilities
lb - object librarian
makefd - resident library utility
cnm - object utility
ord - object utility
diff - comparison utility
grep - pattern search utility

library files

math functions - m.lib
UNIX, ANSI, and general functions - c.lib
Screen Functions - s.lib
Amiga ROM kernel functions - see read.me

header files include:

the complete standard ANSI headers
standard Aztec headers
unbuffered (UNIX) I/O - **fcntl.h**
ROM kernal function prototypes - **functions.h**
Inline ROM kernal function calls - **functions.h**
the complete standard Amiga headers

Library Files for Different Data Definitions

Aztec C supports a number of different data definitions. It would be convenient to have just one set of definitions, but different performance and compatibility requirements make it necessary to provide alternatives. Most non-commercial users and many commercial users will find the default definitions are more than adequate. The data definition in effect is determined by compiler options and by the libraries used to link a program.

The choices marked below with (!) usually produce the smallest and/or fastest code. The items marked with (d) are the system defaults.

The guidelines for choosing system defaults are to provide the configuration that requires the least amount of work and knowledge.

The basic variations for data definitions are:**int definition**

16 bit (!)
32 bit (d)

data and code reference definition

small code and data (!d)
large code and data

floating point definition

Manx IEEE Floating Point (d)
Amiga IEEE Floating Point
Motorola Fast Floating Point (!)
68881 Floating Point (!)

Notice that 16 bit ints produce smaller faster code, but 32 bit ints are the system default. Why not just default to 16 bit ints? The reason is that Amiga supplied routines are based on 32 bit ints. The use of 16 bit ints requires special attention when calling Amiga routines. Either the Amiga

parameters that require 32 bit data items must be declared as longs or the **functions.h** header file must be included. Likewise, header files are required for calling some Manx Aztec routines if 16 bit ints are used. To make Aztec C easier to use, the least complicated configuration is offered as the default. Experienced users who want the increased speed of 16 bit ints or some other system enhancements can provide the necessary option to the compiler, set up different libraries for linking, and include the necessary header (.h) files in their programs.

The three basic Aztec libraries are **c.lib**, **m.lib**, and **s.lib**. These libraries assume 32 bit ints, small code and data, and Manx IEEE floating point.

The complete set of **c.lib**s is

c.lib	32 bit ints	small code and data
c16.lib	16 bit ints	small code and data
cl.lib	32 bit ints	large code and data
cl16.lib	16 bit ints	large code and data

The complete set of **s.lib**s is: **s.lib**, **s16.lib**, **sl.lib**, and **sl16.lib**.

The four basic floating point libraries are:

m.lib	Manx IEEE Float
ma.lib	Amiga IEEE Float
mf.lib	Motorola Fast Floating Point (FFP)
m8.lib	68881 Float

Each of these libraries have four variations:

m.lib , ma.lib , mf.lib , m8.lib	32 bit ints and small code/data
ml.lib , mal.lib , mfl.lib , m8l.lib	32 bit ints and large code/data
m16.lib , ma16.lib , mf16.lib , m816.lib	16 bit ints and smallcode/ data
ml16.lib , mal16.lib , mfl16.lib , and m8l16.lib	16 bit ints and large code/ data

So, as an example, if **68881** floats were used with 16 bit ints in large code and data mode, the linker libraries would be **cl16.lib** and **m8l16.lib**. The compiler options would be: **-ps**, **-mcd**, and **-f8**. If screen functions were used then the **sl16.lib** library would also be needed.

Environment Variables

Manx Aztec C supports four environment variables. These variables are set using the Manx `set` command. The Amiga CLI environment also has a `set` command. The two commands at the time of this writing are not compatible. One way of avoiding confusion would be to rename the Aztec `set` command to `azset` or some other unique name. The `read.me` file may contain more recent information on this issue.

More detailed information on environment variables will be found in the *Aztec C Reference Manual* and in the *Aztec C Unitools Manual*.

The four environment variables supported by Aztec C and their associated commands are as follows:

- CCOPTS - default compiler options (compiler - cc)
- CCEDIT - quick fix command line (compiler - cc editor - z)
- INCLUDE - search path(s) for header files (compiler - cc)
- CLIB - search path(s) for libraries (linker - ln)

Resident Libraries

Amiga Workbench

The Amiga Workbench routines are all accessed through the resident library mechanism. Unlike other library routines that are linked in with a program, resident library routines are accessed through a dynamic mechanism at run time. There are two ways that a program written in Manx Aztec C can reference the Workbench resident library routines, by the use of glue routines or through inline function calls.

A complete set of glue routines for the Amiga Workbench is included in each of the `c.lib` libraries. So, when an Amiga Workbench routine is called the appropriate glue routine is executed to set up the correct environment for the call to the resident routine.

A faster way to access Amiga Workbench routines is by using inline calls. If the header file `functions.h` is included in a program by coding,

```
#include <functions.h>
```

then, instead of generating code to call a glue routine, the compiler will generate an inline direct call to the Amiga Workbench routine.

For more information on resident library support see the *Aztec C Reference Manual Compiler Chapter*. For information on Workbench routines see the *Aztec C Library Manual*.

User Created Resident Libraries

You can create your own resident library routines. Instructions for creating them can be found in the Compiler Chapter of the *Aztec C Reference Manual*.

#pragmas

The creation of inline calls to resident library routines is implemented by use of the ANSI feature, **#pragma**. The Compiler Chapter of the *Aztec C Reference Manual* discusses the pragmas provided as part of Aztec C. It includes pragmas for resident library calls, registerized function calls, and interrupt handler definition.

Pre-compiled Headers

Header files contain definitions for structures, prototypes, and pragmas. There are also **#define**, **typedef**, and **enum** statements. Aztec C headers contain every kind of C statement except those that produce code.

It is typical that a very small percentage of a header file is used by any one program. The header **functions.h**, for example, contains information for over 500 Workbench calls, but most programs use only a small fraction of them. It is highly convenient, however, to put all of the calls into one header file. The penalty is that every time **functions.h** is included the entire contents is processed by the compiler. To maintain the convenience of putting everything into one file but to reduce the overhead of compilation, Manx Aztec C provides for pre-compiled header files.

To make a pre-compiled header file, create a source file with a **#include** statement for each header to be pre-compiled. The compiler is then invoked with the **-hi** option. It is important that any options that effect data definition also be specified. The data definitions of a pre-compiled header must match the data definition for any compilation that uses it. The compiler processes the headers and writes the compiled table information out to a file. Later this file is read in by the compiler by specification of the **-hi** option. Since all of the information is in compiled form, the overhead is simply that of the I/O to read the data. Pre-compiled headers can save a significant amount of time.

A full description of pre-compiled headers can be found in the Compiler Chapter of the *Aztec C Reference Manual*.

Debuggers - db sdb sdbf

Aztec C for the Amiga has three debuggers, **db**, **sdb**, and **sdbf**. **db** is an assembly language debugger and is described in the *Aztec C Reference Manual*. **sdb** and **sdbf** are C source level debuggers. **sdb** is described in the *Aztec C Source Debugger Manual*. **sdbf** is identical to **sdb** except that it supports Motorola Fast Floating Point.

Third Party Software

There are a number of third party software packages, such as Power Windows and Amiga Lint, that extend the capabilities of Aztec C. Call the Manx sales department for a complete list and current prices of third party software.

Freeware, Shareware, and Bulletin Boards

There is a large body of public domain software for the Amiga. Some freeware/shareware is included on the Manx Aztec Distribution disks. Some public domain software can be found on the Manx bulletin board.

Another good place to look is BIX. Information on joining BIX can be found in *BYTE* magazine.

The Compuserve Amiga conference is accessed by entering,

```
go amigatech
```

At the time of this writing, *Amazing Computing* was publishing a comprehensive list of the AMICUS and Fred Fish disks in each issue. Copies of the disks are provided for a nominal fee. Their number is 1-800-345-3360.

Another source of public domain software would be your local Amiga User Group.

Multiple Segments and Transient Overlays

Manx Aztec C supports the creation of multiple segments. Multiple segments can be scatter loaded. Scatter loading gets around memory fragmentation problems. Multiple segments can also be dynamically loaded as transient overlays. Transient overlays allow very large programs to run in a small memory space. More detailed information on this feature can be found in the Technical Information and Linker Chapters in the *Aztec C Reference Manual*.

Mixed C and Assembly Code

Aztec C offers three levels of assembly support: inline, intermediate, and separate.

Inline assembly support allows 680x0 assembly statements to be mixed directly with C code. The general form for inline assembly is:

c code

#asm

assembly code

#endasm

c code

The output of the Aztec C compiler is assembly code. Usually the assembler is invoked quietly by the compiler, and when the assembler finishes the intermediate assembly code is deleted. You can, however, save the intermediate assembly, modify the code, and invoke the assembler directly.

Entire assembly level source files can be created, separately assembled, and then combined with compiled C modules.

More detailed information on this topic can be found in the *Aztec C Reference Manual* in the Assembler, Compiler, Linker, and Technical Information Chapters.

QuikFix

QuikFix is a new facility with version 5.0 of Aztec C. QuikFix is an interactive interface between the Aztec C compiler and the Z editor, or any

editor that supports AREXX (see the **read.me** file). The compiler is set in QuikFix mode by specifying the **-qf** option. The command that is activated if errors are encountered is defined by the CCEDIT environment variable.

QuikFix Compiler Options

The **-qf** option tells the compiler to activate the QuikFix environment. For example:

```
cc -qf hello
```

will compile **hello.c** and, if there are any errors, execute the CCEDIT command string.

When QuikFix is activated, the compiler writes specially formatted error messages to a file called:

AztecC.Err

If errors are encountered during compilation, the compiler will execute the contents of the CCEDIT environment variable as an AmigaOS command. When the **z** command finishes with a zero value, the compiler recompiles the program using the original options. If a non-zero code is returned to the compiler the compiler terminates with an error.

A convenient way to use QuikFix is to specify the **-qf** option in the CCOPTS environment variable. For instance, if you enter:

```
set CCOPTS=-qf
```

in the CLI environment, everytime the compiler is run it will run with the **-qf** option appended to the beginning of the option list.

QuikFix Z Editor Options and Commands

The basic CCEDIT specification to invoke **z** as part of the QuikFix facility is,

```
set "CCEDIT=z -e"
```

The quotes are mandatory. This string tells the compiler to invoke **z** if the **-qf** option was specified. The **-e** option tells **z** that it was invoked as part of the QuikFix facility and, more specifically, to issue the **:cf** command. The **:cf** command causes **z** to read the **AztecC.Err** file. The **AztecC.Err** file has information that indicates the appropriate source file. After reading this

source file, **z** positions the cursor to the line of the first error. At this point, **z** will also display the compiler error message associated with this line.

Three **z** commands are provided to move from error line to error line. They are,

:cp	go to previous error
:cc	redisplay current error
:cn	got to next error

The **Z Editor Chapter** in the *Aztec C UniTools Manual* describes how to assign function keys for **z** via the **z.opt** file. Assigning '**:cp^M**', '**:cc^M**', '**:cn^M**' to three different function keys can save some time in going through files with a large number of errors.

If lines are added to or deleted from the source file, it will effect the error positioning commands. It is a good idea to review the lines in error before making changes and, if it is practical, to make corrections working backwards so that lines added and deleted do not affect the error line positioning commands.

The editor will only process 25 errors at a time. the :cf command will read in the next 25 errors.

If **z** is exited via :q or :wq the compiler will recompile the modified code using the original options. If **z** is exited via :cq then the compiler will terminate with an error status.

Batch Processing with QuikFix

If the compiler is invoked with the **-wq** option, the **AztecC.Err** file will be appended instead of rewritten. This allows multiple compilations to be run with the errors from each written into one **AztecC.Err** file. With the **-wq** option, a command specified through CCEDIT is *not* executed.

By invoking **z** and then issuing the :cf command repetitively you can step from file to file correcting errors. You do not need to know the names of the source files.

Invoking **z** with the **-e** option is equivalent to invoking **z** and immediately issuing the :cf command.

Make and QuikFix

One of the great features of QuikFix is that it can be used in combination with the **make** utility.

To use QuikFix interactively with **make** set up CCOPTS to include the **-qf** option or specify **-qf** with the **make** CFLAGS macro. Then set the CCEDIT environment variable to "CCEDIT=z -e". Then when you use **make**, z will be invoked if there are any source errors during compilation. If the error is corrected, the compiler will recompile the source and **make** will then proceed to any additional files that need to be compiled.

Normally **make** terminates if there is an error in compilation. With QuikFix **make** will only terminate if the z editor is exited with the :cq command.

Further Information on QuikFix

The QuikFix options are documented in the **Z Editor** Chapter of the *Aztec C Unitools Manual* and in the **Compiler** Chapter of the *Aztec C Reference Manual*.

System Performance and Compatibility

The following is a brief discussion of the most significant Aztec C options that effect system performance.

Running the compiler in small code and small data mode in combination with 16 bit ints, produces smaller faster code. The compiler options to produce this result are,

-mcd -ps

The libraries to specify at link time are **c16.lib**, **m16.lib**, and **s16lib**.

When 16 bit ints are used, the **functions.h** header file should be included if Amiga Workbench routines are called. For other library calls, the *Aztec C Library Manual* should be consulted. This manual specifies the header file (.h) that should be included for each library routine. Each header covers multiple routines. It is suggested that you use pre-compiled headers. Pre-compiled headers are discussed earlier in this chapter and in the **Compiler** Chapter in the *Aztec C Reference Manual*.

To produce even smaller faster code add the **-so** option,

-mcd -ps -so

The compiler option, **-so**, causes the compiler to perform maximum code optimization. It is sometimes difficult to debug with this option on since code is rearranged. Some users will run with this option off until the last stage of development. Others run with it on as a matter of course.

To produce faster floating point code, Motorola Fast Floating Point (FFP) or 68881 floating point options can be used. With Motorola FFP format, floats and doubles are 32 bits as opposed to the 64 bits for IEEE format. Thus, the precision is less, but for many applications it is sufficient. The compiler option,

-ff

must be added to the compiler option list for FFP and the **mf.lib** libraries are used for linking. Make sure that you use the right **mf** library. If 16 bit ints are used, for instance, it is **mf16.lib**. To use the 68881 floating point routines, the target machine must have a 68881 co-processor, the compiler option is **-f8**, and the library is one in the **m8** family. The different floating point libraries are discussed earlier in this chapter.

If you are including a large header file in your program, or a large number of smaller header files, compilation time can be speeded up by precompiling some header files. Pre-compiled headers are discussed in the **Overview** Section of this chapter.

The fastest executing software floating point in this package is the Motorola Fast Floating point. If you have a 68881 chip then the fastest floating point is the 68881 floating point.

Making a Working Copy of Aztec C

The #1 distribution disk has an installation procedure that is executed by placing the disk in drive **df0:** and entering,

execute df0:install

This is the preferred installation method.

The following is provided for those who may want to customize the installation beyond what is possible with the install procedure.

1. Make sure that the write protect tabs on the master disks are in the no-write position.

Turn on your machine. If you own an Amiga 1000, insert the KickStart disk and wait until it displays the Insert WorkBench screen, and then insert your standard Workbench boot disk. If you do not own an Amiga 1000, the Kickstart disk is not required, and you can boot with just your Workbench disk.

2. From the Workbench, copy your **sys1:** disk to a blank formatted work disk, then store your original **sys1:** disk in a safe place.
3. You should now organize your disks in a manner which will allow you to comfortably develop applications. How you organize your disks will depend primarily on whether you have one or two disk drives, and also on the amount of memory that is available.

A. Two drives and 1 MB or more of memory available.

The #1 distribution disk has an installation procedure that is executed by placing the disk in drive **df0:** and entering,

execute df0:install

This is the preferred installation method. The following is provided for those who may want to customize the installation beyond what is possible with the install procedure.

Your best approach to setting up this type of system would be to have two working disks which you regularly boot from for C development, and then a third disk which is used to hold your actual C programs.

- Your first disk, which would reside in **df0:** at all times, would be a simple copy of the **sys1:** disk.
- The second disk would contain all utilities, fonts, libraries, etc., that you want to load into the RAM disk.

This may require copying certain directories from your **sys1:** disk to your second working disk. In addition, you should copy certain CLI commands that are not provided on the Aztec disks, such as Format and Diskcopy, from your own Workbench disk to either the first or second boot disk.

For example, if you wanted to have your Amiga libraries, Aztec libraries, and CLI commands loaded into RAM, the portion of your Startup-Sequence that would accomplish this would be:

```
copy df1:libs ram: NUL
assign LIBS: ram:
copy df1:c ram: NUL
path ram:
copy df1:lib ram:
set CLIB=ram:
```

All utilities, include files, etc., that you do not want to load into RAM (or that will not fit in RAM) should reside on the disk which will be in **df0:**. After booting, you can replace the **df1:** disk with your Aztec work disk, which contains all of your C and object files. The basic idea is to have as many commands in RAM as possible, so that both **df0:** and **df1:** can be used for disk copying, C development, etc., instead of for holding system fonts and commands.

B. Two drives and less than 1MB of memory available.

The #1 distribution disk has an installation procedure that is executed by placing the disk in drive **df0:** and entering,

```
execute df0:install
```

This is the preferred installation method. The following is provided for those who may want to customize the installation beyond what is possible with the install procedure.

If you have two floppy drives available, you should set up your first disk to be the Aztec C development disk, containing all necessary files including the compiler, assembler, etc. The second disk should hold all of your object modules, C source code, etc.

You should first decide which files you will need on disk 1. The directories you will need to modify include the C directory, bin directory, devs directory, lib directory, and libs directory.

- In the C directory, you should delete any CLI commands which you do not plan to use, such as Search and Edit. Be sure not to delete the Run command, as it is required by the CLI. Also, you should copy the Format and Diskcopy commands from your Workbench disk if you need them, since the Aztec distribution diskettes do not include these commands.
- In the devs directory, you should delete any devices which you do not plan to use, such as **narrator.device**, (used for speech synthesis) and **printer.device**.
- In the libs directory, you can delete the **translator.library**, which is used by the speech synthesizer. If you are not using transcendental functions, the **mathtrans.library** can be deleted; if you are not using floating point math in your programs, the **mathieeedoubbas.library** can be deleted. You may also delete **info.library** if you do not plan to use the Workbench in your development work.

Libraries

The C libraries you plan to use will depend upon several factors; only the libraries you intend to use should be included on disk 1. First, if you do not plan to use floating point math, delete the **m.lib** library from disk 1, as it will not be needed. Next, make sure *only* the versions of **c.lib** and **m.lib** that you require (if you need them at all) are on the disk.

The libraries that use large code/large data have the letter "l" in their names, and libraries that use 16 bit integers have a "16" in their names. These two types can also be combined. 32 bit ints or small code/small data libraries have no additional letters. As an example, the large code/large data, 16 bit integer c library is **cl16.lib**. The small code/small data, 32 bit c library is **c.lib**.

There are four types of math libraries, distinguished by the characters included in their names (shown in parenthesis):

- Amiga IEEE floating point emulation (64-bit) libraries (**ma**)
- Motorola Fast Floating Point math (32-bit) libraries (**mf**)
- Manx IEEE floating point functions (64-bit) (**m**)
- The 68881 numeric coprocessor libraries (64-bit) (**m8**)

As an example, the large code/ large data 32 bit int Amiga IEEE emulation library is called **mal.lib**. Note that most of the libraries are contained on the **sys2:** disk in the **lib** directory and should be copied to **sys1:** if necessary.

C. One floppy drive.

The #1 distribution disk has an installation procedure that is executed by placing the disk in drive **df0:** and entering,

execute df0:install

This is the preferred installation method. The following is provided for those who may want to customize the installation beyond what is possible with the **install** procedure.

If you have a one-floppy system, your system will based on a trimmed-down version of the **sys1:** disk.

- From the **devs** directory, delete **narrator.device**, and possibly also **printer.device** if you do not need to use a printer.
- From the **libs** directory delete the **translator.library** file. If you are not using transcendental functions in your programs, you can delete **mathtrans.library**; if you are not using floating point, you can delete the **mathieeedoubbas.library**.
- In the **lib** directory, you should keep only the libraries which you need. See the section **Libraries**, above, for details on the libraries that are available.
- From the **C** directory, many commands can be deleted. Notably, the **Edit** editor and the **search** command should be deleted. The **only command you absolutely cannot delete from the C direc-**

tory is the Run command, which is needed by the CLI. You should also copy the format and diskcopy commands from your existing Workbench disk, as these are not present on the Aztec distribution diskettes.

D. Two disk system quick installation.

The #1 distribution disk has an installation procedure that is executed by placing the disk in drive df0: and entering,

```
execute df0:install
```

This is the preferred installation method. The following is provided for those who may want to customize the installation beyond what is possible with the install procedure.

If you have two floppy drives and wish to get up and running right away, you can accomplish this by making duplicates of the first two Aztec disks (**sys1:** and **sys2:**). You then must delete all unnecessary libraries from the lib directory of **sys2:**, as well as the entire lint directory (assuming you have not bought a commercial lint program). See the section **Libraries**, above, for details on the libraries that are available.

Then boot with the copy of **sys1:** in **df0:** and with the copy of **sys2:** in **df1:**. If you plan to follow the tutorial, you should also create a directory called **Aztec** on the **sys2:** disk using the CLI command **makedir**:

```
cd df1:  
makedir Aztec:
```

This will provide a simple working environment for you to start with. If you find this setup too slow and wish to use the RAM drive, or simply need instructions on how to optimize the basic setup, see the instructions in sections **A** or **B**, above.

Using Aztec C with a Hard Disk

The #1 distribution disk has an installation procedure that is executed by placing the disk in drive **df0:** and entering,

```
execute df0:install
```

This is the preferred installation method. The following is provided for those who may want to customize the installation beyond what is possible with the install procedure.

If you have a hard disk, you should install Aztec C onto the hard disk rather than work from floppies. Hard disk installation is fairly straightforward, and consists of the following steps:

A. Enter the system directory on your hard disk and click on the CLI. If it is not there, you may have to start the CLI from your boot floppy. If the CLI is also not on the boot floppy, you will have to run the **Preferences** program to turn on the CLI. See your *Amiga User's Manual* for details on the **Preferences** program.

B. Ensure that you are at the root directory of your hard disk, create a directory named **Aztec** at the root level, and enter the **Aztec** directory :

```
cd dh0:  
makedir Aztec  
cd Aztec
```

C. Insert your **sys1:** disk into the **df0:** drive and type:

```
makedir bin  
copy df0:bin dh0:Aztec/bin
```

to copy the bin directory from **sys1:** to your hard disk. Next, copy the **include** directory:

```
makedir include  
copy df0:include dh0:aztec/include ALL
```

The ALL option for copy specifies that all subdirectories under the **include** directory should be recreated on the destination. You should now copy a program called **set** from the C directory on **sys1:** to the logical **c:** device:

```
copy df0:c/set c:
```

The **set** command will be used later in conjunction with the Startup-Sequence.

Now copy the **lib** directory:

```
makedir lib  
copy df0:lib dh0:aztec/lib
```

D. The **sys1:** disk has now been successfully copied. Remove **sys1:** and put **sys2:** into **df1**. Copy the **bin** and **lib** directories from the **sys2:** disk in the same manner that you did for **sys1:**; but do not create any new directories.

E. Remove the **sys2:** disk and insert the **sys3:** disk. Copy its **bin** and **lib** folders as you did for the **sys2:** disk.

F. You now have to modify your Startup-Sequence so that it properly sets up your system for use with the Aztec C system. If you boot your hard disk system from a floppy, you should insert that floppy now and type:

```
cd df0:s
```

If you have an autoboot drive, simply type:

```
cd s:
```

G. You should now enter your editor of choice and load in the file Startup-Sequence. This is a file which is executed when you boot your machine; it performs general setup work. At the end of the file you should add the following:

```
path add dh0:Aztec/bin  
set CLIB=dh0:Aztec/lib  
set INCLUDE=dh0:Aztec/include
```

If You Have A Problem

If your Aztec C system is not functioning, repeat the installation procedures for your particular hardware/memory setup, paying close attention to each step.

If your Aztec C system is still not functioning, contact Technical Support. See Chapter 4, **Technical Support**, for more information.

TUTORIAL

3

Chapter 3 - Tutorial

This chapter presents a basic tutorial of Aztec C for the Amiga. For this tutorial, it is assumed that you have already followed all of the procedures outlined in Chapter 2, **Getting Started**, and that you are now ready to begin using Aztec C.

This tutorial is meant only to present you with the basics of using the Aztec system; you will be referred to other chapters that will give you more detailed information about the topics discussed here.

AZTEC
SHELL
TUTORIAL

Development Environments

With Aztec C, the default, or boot, environment is the CLI, however there are several public domain shells available which you may want to use in place of CLI, which will work fine with Aztec C.

WORKING THROUGH THE TUTORIAL

This tutorial introduces the CLI and briefly describes some of the more important commands you need to know, including how to organize and move around among your files; how to enter a program; how to compile, assemble, and link your programs; how to set environment variables; and how to take advantage of the Amiga RAM disk.

Using the CLI

STARTING AZTEC C

A. Using floppy disks:

If your machine is not on, turn it on. If you have an Amiga 1000, insert the KickStart disk and wait until it displays the Insert WorkBench screen, but do not insert the WorkBench disk.

If your machine is already on, you must perform a warm boot by holding down the control key and pressing the two Amiga keys simultaneously. After the boot, you should see the Insert WorkBench screen, but do not insert the WorkBench disk.

Instead of your WorkBench disk, insert the Aztec C disk labeled number 1. (Use your working copy--not the original disk!) This disk boots directly into the CLI and will display the Aztec startup message.

B. Using a hard disk:

Boot your hard disk in the usual manner.

- If your normal boot places you into CLI, type

`cd dh0:aztec`

to invoke Aztec C.

- If your normal boot does *not* place you into CLI, then you must go into the system directory and look for the CLI icon.

If there is a CLI icon in your system directory, then double click on it to enter CLI.

If there is *not* a CLI icon in your system directory, exit the system directory, which will place you back at the root level. Double click on the Preferences icon, turn on CLI, save, then exit Preferences. Go into the system directory; you should now see the CLI icon. Double click on this icon to enter CLI.

Once you are in CLI, type

cd dh0:aztec

to invoke Aztec C. Refer to your Amiga documentation if you have any problems with the above procedure.

C. Startup Message

When booting from your first floppy disk, you will see the startup message:

```
Welcome to the Wonderful World of Aztec C!!  
Version X.X  
XX/XX/XX
```

The current date and time is requested:

```
Date (MM/DD/YY) ?  
Time (HH:MM:SS) ?
```

Respond to each in the specified format.

You should see a prompt that looks like:

```
1 >
```

This is the standard CLI prompt; it indicates that the CLI is waiting for a command.

LOOKING AROUND

To begin, type the following command to the CLI:

info

This command displays information about the disks that are currently in the disk drives. It tells you exactly how much space is free on each of your disks.

The Amiga supports what is known as a "hierarchical" file structure that is used to organize files on the disk. An analogy may help to explain this.

Think of a disk as a closet where you want to store different sorts of items. Each disk you have can be thought of as a different closet. If you stuffed all the items you have into this closet, it would become cluttered, making it difficult to find individual items.

So to be more organized, you can let your closet have different sections inside. Now you can have a section labeled "sports" and another labeled "clothes". You can still save other items in this closet as well.

If you go into the "sports" section of your closet, you can have all your sports equipment there, or you can have some more sections *inside* the "sports" section that are labeled "baseball," "football," and "tennis." This could go on for quite a bit, but you should get the idea. With the Amiga, you can create exactly this kind of organization on your disks.

The name you use when talking about a "section" on the disk is a "directory." Type the command:

dir

This is short for DIRectory and displays the contents of the current directory (or "section") that you are in. You will notice that the first set of files displayed is followed by the word **dir**. This is short for "directory" and indicates that there are some additional directories inside the current directory.

To go into one of these directories, type:

cd include

where **include** is the name of one of the files in your previous display that was followed by the word **dir**. The **cd** command is short for "Change Directory." Now you are in the **include** directory and another **dir** command will show you its contents—probably some more directories and some more items.

If you are not sure where you are at any time, type:

cd

to find your current directory, or location. The first name will be master directory that you are in—usually **df0:** or **df1:**, depending upon which drive you are using. This is followed by the name of each subdirectory that you went through to reach the current directory. Each name is separated from the preceding name by a "/" character.

You can move among directories by using the **cd** command and typing the full name of the directory you want. For example, to go into the **df0:** drive's **c** directory with the first **dir** command, type:

cd df0:c

If the directory does not exist, an error message is displayed.

Once you are in the **c** directory, try the **dir** command. You will notice that some of the files listed have familiar names like **cd**, **dir**, **format** and **diskcopy**. This is the command directory and is where the CLI gets all the commands that it executes.

To move again, type:

```
cd df0:include/exec
```

This places you into the **exec** directory, which is in the **include** directory, which is on the **df0:** disk.

Now, instead of **dir** type:

```
list
```

This command is like **dir**, but it gives more information about each file and directory in the current directory.

A similar command:

```
type errno.h
```

displays the contents of the file **errno.h** on the screen of the Amiga.

The commands **info**, **cd**, **dir**, **list** and **type** are the most often used commands when moving and looking around in the CLI. Try using these commands to look in other directories as well.

WORKING WITH FILES

Move to the top or **ROOT** directory by typing:

```
cd df0:
```

Now, make a directory for your files, to keep them out of the way. Type:

```
makedir test
```

which creates a new directory named **test** in the current directory. Go into **test** by typing:

```
cd test
```

To create a simple program using the **copy** command, type:

```
copy * to hello
```

The "*" tells the copy command to get the source from the keyboard and the **to** says to put what it gets into the file **hello**.

The disk whirs and the cursor waits for input without a prompt. Now, type the following short program, exactly as it is written here:

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

To tell the **copy** command you are done, hold down the **ctrl** key and press the **** key. You should see the CLI prompt once again.

To make sure the file is correct, use the **type** command again to display the file to the screen. Proof the text that is displayed against the above program; they should be identical.

If you now type **dir**, you will see the file **hello**. Since most C program source files are distinguished by a file name extension of **.c**, use the following command to rename the file:

```
rename hello hello.c
```

This changes the name of the file **hello** to **hello.c**.

A much better way to create and modify files is to use a text editor. The Amiga computer comes with two editors which are described in the *AmigaDOS User's Manual*, **ed** and **edit**. **Ed** is generally considered a much better editor than **edit**, since **ed** is screen edited, while **edit** is line-oriented, which most people find cumbersome. In addition, Aztec C includes a powerful editor named **Z**, which is similar to the UNIX **vi**-editor.

COMPILING, ASSEMBLING, AND LINKING

To make an executable version of your file that you can run, type the following to compile and automatically assemble this program:

```
cc hello.c
```

This invokes the Aztec C compiler to compile the file **hello.c**; it will also automatically assemble the file as well. Do not worry if you receive an

error: illegal character 0x1b (this is caused by the ctrl-\ sequence we used earlier), as it does not affect the code generated.

Do a **dir** command and you will see that a new file, **hello.o**, has been created. This is the object file created by the assembler.

To turn **hello.o** into an executable file, you need to link it with the **c.lib** library. Type:

```
ln hello.o -lc
```

This invokes the linker to link the **hello.o** module with the standard C library and place the output in the file **hello**. Notice how much disk activity takes place during the link phase. The linker is taking all the pieces from the library that it needs and is writing them out to the executable file.

To execute the program, type:

```
hello
```

The words

```
hello world!!!
```

should appear on your screen, followed by a prompt.

To get rid of the **.o** file, type:

```
delete hello.o
```

which deletes the file.

ENVIRONMENT VARIABLES

When the linker links with the libraries, or when the compiler includes header files, you have to tell them where to find them. The easiest way is to use something called an environment variable.

This is a variable with a name whose value is a string and that can be located by any program. Environment variables are created with the **set** command. To see what environment variables currently exist, type:

```
set
```

and the current environment will be displayed.

Notice that two items are defined, CLIB and INCLUDE. These were created by a command in a file whose full name is df0:s/Startup-Sequence. The command in the file looks like:

```
set CLIB=df0:lib/INCLUDE=df0:include
```

The linker uses the CLIB variable to know where to look for the libraries, while the compiler and assembler use the INCLUDE variable to find header files.

THE RAM DISK

When you linked the "hello" program, the linker had to look all over the disk for the parts of the library that it needed and then put them together to make the program. This is why linking took a while. However, you can speed up linking by speeding up the seeking process.

This is accomplished nicely by using a very fast disk or by using a RAM disk. The Amiga computer comes with a built-in RAM disk that lies dormant until you want to use it. Since the RAM disk uses up memory that is also needed for programs to run, you need at least 512K of memory to use it.

To make the link run faster, you first need to copy the library over to the RAM disk. Type:

```
copy df0:lib/c.lib ram:
```

Now, you need to tell the linker where to look, so type:

```
set CLIB=ram:
```

which changes the environment variable used by the linker.

Now if you type:

```
ln hello.o -lc
```

there should be a lot less disk activity, and the linking should be much faster.

Summary

This chapter has described only a few of the many features of the Aztec C development system. From here, you should go on to read the **Compiler**, **Assembler**, and **Linker** chapters of your *Aztec C Reference Manual*.

At some point, you should read the entire *Reference Manual*, as well as the *Aztec C Library Manual*. If you are not already familiar with the AmigaDOS CLI commands and options, it would also be well worth your while to read the *AmigaDOS User's Manual*.

Once you are accustomed to using Aztec C, you can start writing programs that access the special features of the Amiga. These features are discussed briefly in the **Compiler** chapter of your *Aztec C Reference Manual* and in the **Library Functions** chapter of your *Aztec C Library Manual*. We recommend, also, that you purchase the *ROM Kernel* and the *Intuition* manuals, as these books describe all the functions in detail and provide examples of their use.

For more information on these and other Amiga-related books, see the section "Suggestions For Further Reading" that has been included at the front of this manual.

Now that you have completed the **Tutorial**, you should know enough to create some simple programs using Aztec C. As helpful examples, a number of public domain programs have been collected from various sources and included on your Aztec C disks.

Good luck, and enjoy!

TECHNICAL SUPPORT

4

Chapter 4 - 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 page 4-5 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 page 4-5 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 page 4-5 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 10-12 a.m. and 3-5 p.m. eastern standard time. (Times subject to change without notice; for the correct telephone number, see page 4-5.) 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	(201) 542-1795
Sales (Domestic)	(800) 221-0440
Sales (International)	(201) 542-2121
Updates	(201) 389-0290
FAX:	(201) 542-8386
Bulletin Board: 300/1200 bps (all products)	(201) 542-2793

Note: The above information is subject to change without notice.

MANX PROBLEM REPORT

Date: ____ / ____ / ____

Name: _____

Phone #: 1-(____)-____ - ____

Company : _____

Address : _____

Product

C86-PC ____ C86-CPM86 ____

C68k ____

C68k-Am ____ cII ____ c80 ____

c65-ProDos ____ c65-DOS3.3 ____

cross _____

Version #: _____

Serial #: _____

Op. - sys.: _____

Machine Config.: _____

Description of problem --

Include what has already been attempted to fix the problem and use the reverse side of this sheet if needed.

Index

68020, 1-1
68881, 1-1

A

add-on products, 1-1, 1-3
Amiga 1000, 2-2
Amiga 2000, 2-2
Amiga 2500, 2-2
Amiga 500, 2-2
ANSI standard, 1-6
assembly and C mixed, 2-10
autoboot
 Startup-Sequence, 2-21
Aztec C
 core systems, 1-1
 Developer level, 1-2
 Professional level, 1-2
Aztec C add-on products
 See add-on products

B

backing up distribution disks, 2-2
bin directory, 2-17
branch shortening, 1-4
bulletin boards, 2-9

C

C and assembly mixed, 2-10
C directory, 2-17
C libraries, 2-17
c.lib, 2-17, 3-7
cc command, 3-6
cd command, 3-4
CLI
 default environments, 3-1
 loading into RAM, 2-16

makedir, 2-19
Preferences program, 2-20
prompt, 3-3
run command, 2-17
using CLI, 3-2

CLI commands

- cd, 3-4
- copy, 3-5 - 3-6
- delete, 3-7
- dir, 3-4
- info, 3-3
- list, 3-5
- rename, 3-6
- type, 3-5

CLI icon, 3-2

CLIB environment variable, 3-8

compile, 3-6

components, Aztec C

- debuggers, 2-9
- files, 2-4 - 2-5
- libraries, 2-5 - 2-6
- pragmas, 2-8
- pre-compiled header files, 2-8
- resident libraries, 2-7

copy command, 3-5 - 3-6

- example, 3-8

core systems, 1-1 - 1-2

cross development, 1-3

D

data definitions, 2-5 - 2-6

debuggers, 2-9

delete command, 3-7

Developer System, 1-2

development environments

- choice of, 3-1

device drivers, 1-1

devs directory, 2-17

diff, 1-2

dir command, 3-4

directory
 command, 3-4
 definition, 3-4
 display current, 3-4 - 3-5
Diskcopy command, 2-16
documentation, 1-1, 1-3
 reading order, 1-4 - 1-5
draft-proposed ANSI, 1-6

E

ed, 3-6
edit, 3-6
editor, 3-6
environment variables, 2-7, 3-7
 CCEDIT, 2-11 - 2-13
 CCOPTS, 2-11, 2-13
 CLIB, 3-8
 INCLUDE, 3-8
executable file, 3-7

F

file structure, 3-3
file types, 2-4 - 2-5
floating point libraries, 2-14, 2-17 - 2-18
 16 bit int library, 2-17
 32 bit int library, 2-18
 64 bit int library, 2-18
 68881 coprocessor, 2-18
 Amiga IEEE, 2-18
 large code/large data , 2-17
 Manx IEEE, 2-18
 Motorola Fast Floating Point, 2-18
 small code/small data , 2-17
Format command, 2-16
freeware, 2-9
functions.h, 2-5 - 2-8, 2-13
 -hi compiler option, 2-8

G

grep, 1-2

H

hard disk, using Aztec C with, 2-19

I

include directory, 2-20

INCLUDE environment variable, 3-8

info command, 3-3, 3-5

installation procedure, 1-4, 2-3, 2-14

 hard disk, 2-19 - 2-21

 one floppy drive, 2-18

 two drives + 1 MB memory, 2-15 - 2-16

 two drives + less than 1 MB memory, 2-16 - 2-17

 two drives, quick installation, 2-19

Intuition, 1-1

L

large code/large data, 2-17

lib directory, 2-17

library source, 1-3

libs directory, 2-17

link, 3-7

linking, 3-7 - 3-8

lint directory, 2-19

list command, 3-5

M

m.lib, 2-17

make, 1-2

makedir command, 2-19, 2-21

math libraries, 2-17

 See also floating point libraries

multiple segment support, 2-10

N

notation conventions, 1-5

O

object file, 3-7
overlay support, 2-10

P

pragmas, 2-8
pre-compiled header files
 -hi compiler option, 2-8
Preferences program, 2-20, 3-2
Professional System, 1-2

Q

quick installation, 2-19
QuikFix, 2-10
 -qf compiler option, 2-11
 -wq compiler option, 2-12
AREXX, 2-11
batch processing, 2-12
further information, 2-13
make utility, 2-13
Z editor commands, 2-11 - 2-12
Z editor options, 2-11

R

RAM disk, 2-15, 3-8
 system requirements, 3-8
RAM loading, 2-16
reading order, 1-4
README file, 1-4, 2-2 - 2-3
resident libraries
 Amiga Workbench, 2-7
 user created, 2-8
root directory
 defined, 3-5
Run command, 2-17, 2-19

S

sdb, 1-3, 2-2
set command, Amiga CLI, 2-7
set command, Aztec, 2-7, 2-20, 3-7
shareware, 2-9
small code/small data, 2-17
source level debugger, 1-3, 2-2
 See also sdb
starting Aztec C
 with floppy disks, 3-2
 with hard disk, 3-2
startup message, 3-3
Startup-Sequence, 2-16
system overview, 2-3
system requirements, 2-1 - 2-2, 2-15 - 2-16

T

technical support, 4-1
 bulletin board system, 4-3
 mail-in service, 4-3
 phone hours, 4-4
 phone number, 4-4
 phone support, 4-4
 procedures, 4-2 - 4-3
 required info, 4-1 - 4-2
 updates, 4-3

text editor, 3-6
third party software, 2-9
transcendental functions, 2-17
U

UniTools, 1-2, 1-5
UNIX, 1-1 - 1-2, 3-6

W

write protect tabs, 2-15

Z

z editor, 1-2, 3-6

Aztec C Reference Manual for the Amiga

**Version 5.0
October 1989**

Copyright © 1988, 1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

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

This software package and document are copyrighted ©1988, 1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed,
stored in any retrieval system, or translated into any language without prior
written permission of Manx Software Systems.

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, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh 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

October 1989	Fifth Edition
December 1988	Fourth Edition
May 1988	Third Edition
April 1986	Second Edition
February 1986	First Edition

Table of Contents

Chapter 1 - Overview

Introduction	1-1
ANSIC	1-1

Chapter 2 - Compiler

Using the Compiler	2-1
Input File	2-2
Source Filename Extensions	2-2
The Output Files	2-3
Creating an Object File	2-3
Creating an Assembly Language File	2-4
Searching for #include Files	2-5
Option -i	2-5
#include Search Order	2-6
Pre-compiled #include Files	2-6
Memory Models	2-8
Large Data versus Small Data	2-8
Large Code Versus Small Code	2-9
Selecting a Module's Memory Model	2-12
Libraries	2-12
Multi-module Programs	2-13
Code Segmentation	2-14
Compiler Options	2-14
Option Philosophy	2-14
CCOPTS Environment Variable	2-15
Option List	2-17
Option Descriptions	2-21

Option -3	2-21
Option -5	2-22
Option -c2	2-22
Option -d macro	2-22
Option -fa	2-23
Option -ff	2-23
Option -fm	2-24
Option -f8	2-24
Option -hi filename	2-24
Option -ho filename	2-24
Option -i path	2-25
Option -k	2-25
Option -mb	2-26
Option -mc	2-26
Option -md	2-26
Option -mm	2-26
Option -o filename	2-27
Option -pa	2-27
Option -pb	2-27
Option -pe	2-27
Option -pl	2-27
Option -ps	2-28
Option -pt	2-28
Option -qa	2-28
Option -qf	2-29
Option -qp	2-29
Option -qq	2-29
Option -qs	2-29
Option -qv	2-29
Option -sf	2-29
Option -sn	2-30
Option -sp	2-30
Option -sr	2-30

Option -ss	2-30
Option -su	2-31
Option -wa	2-31
Option -wd	2-31
Option -wn	2-32
Option -wl	2-32
Option -wo	2-32
Option -wp	2-32
Option -wq	2-33
Option -wr	2-33
Option -wu	2-33
Option -ww	2-34
Table Manipulation Options	2-34
Error Handling	2-34
Writing Programs for the Amiga	2-35
#pragma for Amiga Exec Calls	2-35
Calling Resident Libraries	2-36
Creating Resident Libraries	2-38
#pragma for Register Function Calls	2-39
Miscellaneous Notes on Resident Libraries	2-40
Pointer Considerations	2-41
Declare Functions That Return a Pointer	2-41
Declare Function Arguments That Are Pointers	2-42
Pointer Variables And Constants As Arguments On Calls	2-42
Other Considerations	2-43
Internal Storage Of Numeric Data	2-43
Converting Data	2-43
Additional Features	2-44
Line Continuation	2-44
String Concatenation	2-44
In-line Assembly Code	2-44

Chapter 3 - Language Specification

Compatibility Issues	3-1
Compatibility with Other Aztec Compilers	3-1
Keywords	3-2
Operators	3-2
Pre-Processing Directives	3-2
Type Information	3-2
Integers and Long Integers	3-3
Characters	3-4
Structures	3-4
Bitfields	3-5
Enum Constants	3-5
Const and Volatile	3-5
General ANSI-Compatible Features	3-6
Function Prototyping and ANSI Function-Definition	3-6
ANSI Standard	3-7
Auto Aggregate Initialization	3-10
Wide Strings and Literals	3-10
Structure Assignment and Structure Passing	3-11
Trigraphs	3-11
Escape Sequences	3-12
Long Character Constants	3-12
Register Variables	3-13
Sizeof and Size_t	3-13
Symbol Names	3-13
Preprocessor Features	3-13
Predefined Symbols	3-13

String Concatenation	3-15
Line Continuation	3-15

Chapter 4 - Assembler

Operating Instructions	4-2
EXECUTION ENVIRONMENT	4-2
Processor Support	4-2
INPUT FILE	4-2
OBJECT CODE FILE	4-3
LISTING FILE	4-3
OPTIMIZATIONS	4-3
SEARCHING FOR INCLUDE FILES	4-4
Option <i>-i</i>	4-4
INCLUDE Environment Variable	4-5
Include Search Order	4-5
MODULE MEMORY MODEL	4-5
Assembler Options	4-6
SUMMARY OF OPTIONS	4-6
DESCRIPTION OF OPTIONS	4-7
Option <i>-c</i>	4-7
Option <i>-d</i>	4-7
Option <i>-o</i> filename	4-7
Option <i>-i</i>	4-7
Option <i>-l</i>	4-7
Source Program Structure	4-8
COMMENTS	4-8
EXECUTABLE INSTRUCTIONS	4-8
LABELS	4-8
OPERATIONS	4-8

OPERANDS	4-9
DIRECTIVES	4-10
blanks	4-10
clist and nclist	4-10
cnop	4-11
cseg	4-11
dseg	4-11
dc - Define Constant	4-12
dcb - Define Constant Block	4-12
ds - Define Storage	4-13
entry	4-13
end	4-13
equ	4-13
equir	4-14
even	4-14
fail	4-14
far code	4-14
far data	4-14
freg	4-14
ifc and ifnc	4-15
ifd and ifnd	4-15
if, else, and endc	4-15
near code	4-16
near data	4-16
other ifs	4-16
include	4-16
global and bss	4-16
list and nolist	4-17
mlist and nomlist	4-17
machine	4-17
macro and endm	4-17
mc68851	4-18
mc68881	4-18

mexit	4-18
public	4-19
reg	4-19
section	4-19
set	4-19
ttl	4-20
xdef and xref	4-20
MACRO CALLS	4-20
Interfacing With C	4-21
REGISTER CONVENTIONS	4-21
ARGUMENTS TO SUBROUTINES	4-21
Variable Names	4-21
Returning from a C Function	4-22

Chapter 5 - Linker

Introduction to Linking	5-2
LINKING Hello.o	5-2
THE LINKING PROCESS	5-2
Libraries	5-3
EXAMPLE	5-4
The Order of Library Modules	5-5
Using the Linker	5-7
STARTING THE LINKER	5-7
INPUT FILES	5-8
EXECUTABLE FILES	5-8
LIBRARIES	5-9
SUMMARY OF LINKER OPTIONS	5-10
DETAILED DESCRIPTION OF OPTIONS	5-11
Option -l	5-11

Option -f	5-12
Option +l	5-12
Option -m	5-13
Option -o	5-13
Option +o[i]	5-13
Option +s, +ss, +sss	5-14
Option -t	5-14
Option -w	5-14
Option -v	5-14
Options +c and +f	5-15
Option -g	5-16

Chapter 6 - Utilities

arcv	6-2
adump	6-3
cmp	6-4
cnm	6-5
Options	6-6
Symbol Format	6-7
Symbol Types	6-7
ab	6-8
pg	6-8
dt	6-8
un	6-8
bs	6-9
Gl	6-9
du	6-10
hd	6-11

lb	6-12
Options Argument	6-12
FUNCTION CODE OPTIONS	6-12
QUALIFIER OPTIONS	6-13
The mod Argument	6-13
Reading Arguments from Another File	6-13
Basic Features of lb	6-14
HOW TO CREATE A LIBRARY	6-14
HOW TO LIST NAMES OF MODULES IN A LIBRARY ..	6-15
HOW MODULES GET THEIR NAMES	6-15
ORDER OF MODULES IN A LIBRARY	6-15
GETTING LB ARGUMENTS FROM A FILE	6-16
ADVANCED lb FEATURES	6-17
ADDING MODULES TO A LIBRARY	6-17
ADDING MODULES BEFORE AN EXISTING MODULE ..	6-18
ADDING MODULES AT THE BEGINNING OR END OF A LIBRARY	6-18
MOVING MODULES IN A LIBRARY	6-19
DELETING MODULES	6-19
REPLACING MODULES	6-20
UNIQUENESS	6-20
EXTRACTING MODULES	6-21
THE VERBOSE OPTION	6-21
SILENCE OPTION	6-22
REBUILDING A LIBRARY	6-22
DEFINING THE DEFAULT MODULE EXTENSION ..	6-22
HELP	6-22
ls	6-23
mkarcv	6-25
obd	6-26

ord	6-27
set	6-28
Displaying and Setting Environment Variables	6-28
setdate	6-29

Chapter 7 - Debugger

Requirements	7-1
Preview	7-1
Overview	7-2
BASIC COMMANDS	7-2
NAMES	7-3
Code and Data symbols.	7-3
Operator usage of names.	7-3
LOADING PROGRAMS AND SYMBOLS	7-3
BREAKPOINTS	7-4
MEMORY-CHANGE BREAKPOINTS	7-5
TRACE MODE	7-6
BACKTRACING	7-6
MACROS	7-6
OTHER FEATURES	7-7
Using DB	7-7
STARTING DB	7-7
TERMINATION	7-8
SUPPORT FOR SCATTER-LOADED PROGRAMS	7-8
SUPPORT FOR SEGMENTED PROGRAMS (OVERLAYS) ...	7-8
INPUT/OUTPUT	7-9
COMMANDS	7-9

Definitions	7-9
EXPR	7-9
TERM	7-10
ADDR	7-13
RANGE	7-13
CMDLIST	7-14
COMMAND DESCRIPTIONS	7-14
The Amiga Commands	7-14
add	7-14
adi	7-14
adl	7-14
adp	7-14
adr	7-14
ai	7-15
ak	7-15
al	7-16
aL	7-16
an	7-17
aN	7-17
am	7-17
ap	7-17
aP	7-17
aq	7-18
ar	7-19
as	7-19
aS	7-19
at	7-19
The Breakpoint Commands	7-20
bb	7-20
bw	7-20
bl	7-20
bc	7-21
bC	7-21

bd	7-21
bh	7-22
bq	7-22
br	7-23
bs	7-23
bt	7-24
bT	7-24
bu	7-24
The Clear Commands	7-25
cs	7-25
The Display Commands	7-26
db	7-26
dw	7-26
dl	7-26
d	7-26
dc	7-26
dd	7-27
dg	7-27
ds	7-27
The Go commands	7-28
g	7-28
G	7-28
The Load Commands	7-29
ls	7-29
The Memory Modification Commands	7-29
ma	7-29
mb	7-30
mw	7-30
ml	7-30
mc	7-30
mf	7-31
mm	7-31
ms	7-31

The Radix Command	7-32
n	7-32
The "Print" Command	7-32
p	7-32
A complete list of desc_codes	7-37
The Quit command	7-39
q	7-39
The Register command	7-40
r	7-40
The Single Step commands	7-40
s	7-40
S	7-40
t	7-40
T	7-40
The Unassemble commands	7-41
u	7-41
U	7-41
The Variable commands	7-42
v	7-42
V	7-42
The Macro command	7-42
x	7-42
The Display Expression Command	7-43
=	7-43
The Redirect Input/Output Commands	7-43
<	7-43
>	7-43
>>	7-43
The Help command	7-44
?	7-44
Command Summary	7-45
AMIGA COMMANDS	7-45

BREAKPOINT COMMANDS	7-45
CLEAR COMMANDS	7-46
DISPLAY COMMANDS	7-46
GO COMMANDS	7-46
LOAD COMMANDS	7-46
MEMORY MODIFICATION COMMANDS	7-46
RADIX COMMAND	7-46
FORMATTED PRINT COMMANDS	7-46
QUIT COMMAND	7-46
REGISTER COMMAND	7-47
SINGLE STEP COMMANDS	7-47
UNASSEMBLY COMMANDS	7-47
VARIABLE COMMAND	7-47
MACRO COMMAND	7-47
DISPLAY EXPRESSION	7-47
INPUT/OUTPUT COMMANDS	7-47
HELP COMMAND	7-47

Chapter 8 - Technical Information

Program Organization	8-2
PLACEMENT OF SEGMENTS OF MEMORY	8-3
SEGMENT SIZE	8-4
Segmentation and Segmented Code	8-5
SEGMENTATION	8-5
Loading a Segment	8-5
Unloading a Segment	8-6
Segload	8-6
Linking and Startup Code	8-7

Segment Construction	8-7
Segment Data Table	8-8
Hunk Data	8-8
SEGMENTED CODE	8-8
Programmer Information	8-8
Global and Static Data	8-9
Operator Information	8-9
Example	8-9
Including Modules from Libraries	8-10
Reselecting Segments	8-10
Object Module Libraries	8-11
Assembly Language Functions	8-11
CONVENTIONS FOR C CALLABLE, ASSEMBLY LANGUAGE FUNCTIONS	8-12
Names of External Functions and Variables	8-12
Function Calls and Returns	8-13
Global Variables	8-14
Register Usage	8-15
Embedded Assembler Source	8-15
Interrupt Service Routines	8-16
Creating SubTasks	8-17
Floating Point Information	8-18
CREATING A FLOATING POINT FORMAT PROGRAM ...	8-19
FLOATING POINT FORMATS	8-19
Building Programs that Run from Workbench	8-19
OVERVIEW	8-19
STEP BY STEP	8-20

Chapter 9 - Error Messages

List of Error Messages	9-2
Compiler Error Messages	9-2
Fatal Compiler Error Messages	9-6
Internal Errors	9-7
Assembler Error Messages	9-7
Opcode Error Messages	9-7
Directive Error Messages	9-8
Syntax Errors	9-9
Linker Error Messages	9-10
Command Line Errors:	9-10
I/O Errors	9-10
Errors in Use of Memory	9-11
Errors Arising From Source Code	9-11
Compiler Error Messages	9-12
Fatal Compiler Error Messages	9-58
Internal Errors	9-62
Assembler Error Messages	9-62
Opcode Error Messages	9-62
Directive Error Messages	9-66
Syntax Errors	9-69
Linker Error Messages	9-69
Explanation of Linker Error Messages	9-70
Command Line Errors	9-70
I/O Errors	9-71
Errors in Use of Memory	9-73
Errors Arising From Source Code	9-74

OVERVIEW

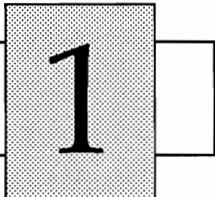
COMPILER

LANGUAGE
SPECIFICATIONS

ASSEMBLER

LINKER

OVERVIEW



1

Chapter 1 - Overview

Introduction

This is the *Aztec C Reference Manual* and is included as a part of your total Aztec C documentation package. See page 1-3 of your *Aztec C User Guide* for a listing of all of the Aztec C configurations and documentation that are available for the Amiga.

The *Aztec C User Guide* showed you how to install your software and how to begin using Aztec C easily and quickly. This *Aztec C Reference Manual* presents more advanced programs and a more thorough understanding of how the Aztec C package works.

The *Aztec C Library Manual* includes an overview, listing, and description of all the library functions that are included with your Aztec C package, and a listing of the Amiga Workbench function names and syntax.

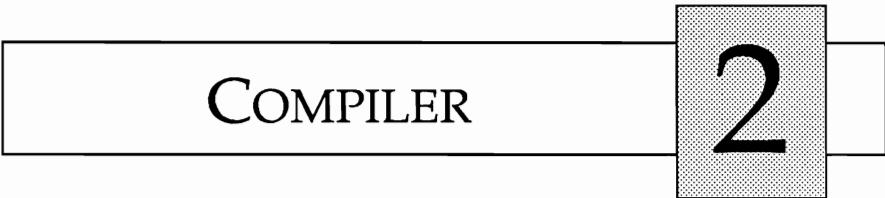
Of course, this *Aztec C Reference Manual* also contains a detailed Table of Contents and an Index.

ANSI C

Aztec C is fully ANSI standard. If you have used Aztec C prior to Version 5.0, you should note that this standardization has caused the compiler and its options to change substantially. Before using Aztec C, you should carefully review the **Compiler** chapter.

Other chapters that have been affected by the ANSI standardization and should be reviewed include **Assembler**, **Utilities**, **Technical Information**, **Error Messages**, and **Style**.

If you are a previous user, you should also note that a new chapter has been added to this manual: **Language Specifications**. This chapter discusses Aztec C's implementation of ANSI C.



Chapter 2 - Compiler

This chapter describes how to use the Manx Aztec C Compiler. The Aztec C compiler is implemented according to the draft proposed ANSI standard. For information on the C language and its use, refer to the section at the beginning of your *Aztec C User Guide*, "Suggestions for Further Reading," which lists several tutorial texts for the C student.

This chapter contains four topics: 1) how to use the compiler, 2) compiler options, 3) error handling, and 4) programmer information, such as the use of register variables and the writing of machine-independent code.

Using the Compiler

To invoke the Aztec C compiler, use the following command:

```
cc [options] filename.c
```

where [options] specifies optional parameters, and *filename.c* 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 *filename.c*, translates them to assembly language, and writes the results to an output file.

If the QuikFix option is enabled (option *-qf*) the compiler on encountering errors will dynamically invoke an editor and interactively help locate

source lines with errors. A windowed display is provided of the associated error message. This facility is described in the *Aztec C User Guide*.

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

To abort a compilation, press the Control-C keys.

Input File

When you execute 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 **dsk2:db/source**, you can compile it by using the following command:

```
cc dsk2:db/source/prog1.c
```

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

```
cc prog1.c
```

And if the current directory is **db**, on the **dsk2:** volume, you could compile the file with the command

```
cc 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

```
cc 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

```
cc prog.prg
```

reads the file **prog.prg** from the current directory as the input to the compiler.

The Output Files

Creating an Object 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 compiler-started assembler is sent to a file whose name is derived from that of the file containing the C source by changing its extension to **.o**. This file is placed in the directory that contains the C source file.

For example, if you started the compiler with the command:

```
cc prog.c
```

the file **prog.o** is created, containing the relocatable object file for the program. You may explicitly specify the name of the object file using the compiler option **-o**. For example, the command

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

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

When the compiler is going to start the assembler automatically, by default it writes the assembly language source to the file **ctmpxxx.xxx**, where **xxx** are numbers chosen such that the file name is unique. The file is placed in the directory defined by the **CCTEMP** environment variable. If **CCTEMP** doesn't exist the file is placed in the current directory.

The **CCTEMP** environment variable can be used to pass the intermediate assembler file to the assembler through a RAM disk. If you are using floppy disks, a RAM disk can definitely save time. If you are using a hard disk, the time savings may be minimal.

If you are interested in the assembler source, but still want the compiler to start the assembler, 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

```
cc -at prog.c
```

compiles and assembles **prog.c**, creating the files **prog.asm** and **prog.o**.

Creating an Assembly Language File

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

```
cc -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

```
cc -a -o ram: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 volume named **ram**:. When option **-a** is used, sub-option **-t** causes the compiler to include the C source statements as comments in the assembly language source. Usually you will want to also specify **-at** to request cc to insert the original C source as comment code when it generates the assembly file. (See the "Compiler Options" section of this chapter for more information.)

Searching for #include Files

By default the Aztec C compiler searches the current directory to locate files specified in **#include** statements. It can also search a user-specified sequence of directories for such files, thus allowing program source files and header files to be contained in different directories.

Compiler option **-i** and the environment variable **INCLUDE** define the directories in which the compiler searches for **#include** files. The compiler automatically searches the current directory for a **#include** file if the following conditions are met:

- (1) the compiler is started without specifying option **-i**,
- (2) There is not an **INCLUDE** environment variable, and
- (3) the **#include** statement does not specify the drive and/or directory containing the file.

If a **#include** statement specifies either the drive or directory, only that location is searched for the file.

Option **-i**

Compiler option **-i** defines a single directory to be searched for the file that was specified in a **#include** statement. The path descriptor follows option

- i. A space between -i and the path descriptor is allowed but not required.

The specification

```
cc -i sys:db/include
```

directs the compiler to search the **sys : db / include** area when looking for an include file.

Multiple -i options can be specified when starting the compiler, if desired, thus defining multiple directories to be searched.

You can also specify where include files are kept, using the INCLUDE environment variable. See your *Aztec C User Guide* for more information on using the INCLUDE environment variable.

#include Search Order

When the compiler encounters a #include statement, it searches directories for the file specified in the statement in the following order:

- if the filename is delimited by double quotes, "*filename*," the current directory is searched.
- if the name is delimited by angle brackets, <*filename*>, the current directory is searched only if no -i options are specified and if the INCLUDE environment variable does not exist.
- directories specified in option -i are searched, in the order listed on the line that started the compiler.
- directories specified in the INCLUDE environment variable are searched, in the order listed.

Pre-compiled #include Files

To shorten compilation time, the compiler supports pre-compiled #include files.

This option only applies to a block of **#include** files that are located in the beginning of a file. For example, if you place a **#undef** between two **#include** file statements, it will not have the desired effect.

To use this feature, you must first create the pre-compiled header file. This is done by compiling the desired header files with **-ho** option. This option tells the compiler not to generate any code but to create symbol table information which it stores in the file whose name was specified after the **-ho** option. This symbol table file can then be included on subsequent compiles by specifying the **-hi** option followed by the name of the symbol table file. The compiler adds into its symbol table the pre-compiled symbol table information. When the compiler encounters an **#include** statement of a header file for which it has pre-compiled symbol table information, it does not have to compile it, even if the include is nested within another include. This can save a considerable amount of time.

Since there are options that cause the compiler to make different assumptions about the size of data items, it is important to use pre-compiled header files that match the data size assumptions of the compilations where they are specified.

Option **-ho** tells the compiler to write its symbol table to a file. The name of the file follows the **-ho** and a separating space as the next argument. For example, you might create a file named **x.c** that consists only of include statements for all the header files that you want pre-compiled. You could then generate a file named **x.dmp** that contains the symbol table information for these header files by entering the following command:

```
cc -ho x.dmp x.c
```

Option **-hi** tells the compiler to read pre-compiled symbol table information from a file and uses the normal include search path. The name of the file follows the **-ho**, with one intervening space. For example, to compile the file **prog.c** that accesses the header files defined in **x.c**, and to have the compiler preload the symbol table information for these files from **x.dmp**, enter the following command:

```
cc -hi x.dmp prog.c
```

The **-hi** option actually initializes the compiler symbol table. For this reason, only one **-hi** is permitted, and any conditionally-defined objects are in-

cluded or not, depending on definitions when the pre-compiled header (.dmp) file was created, and not when it is used.

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, following is a brief description of the sections into which a C-generated program is organized. The sections of a program include:

- **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 and 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, and 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 between large data and small data programs result from this fundamental difference. These other differences are:

1. 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 64K bytes at most of global and static data.
2. For a small data program, an address register must be reserved to point into the middle of the data segment. This is the `a4` register in the Amiga.
3. 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 does not need this address register.
4. It takes more time to load a code segment for a large data program than for a small data program. The reason for this is that the absolute address of the data contained in the data segment is not known until a program is loaded. Therefore, instructions that access data segment data using absolute addresses must be adjusted when the code segment containing the instructions is loaded, whereas instructions that access data segment data in a position-independent way do not need to be adjusted.
5. 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.
6. 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 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, and 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. (small code will be used in most cases, due to the nature of the way it works. Large code is used primarily for absolute function pointers.)

Other differences in large data and small data programs result from this fundamental difference. These other differences are:

1. 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. For example, 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.

2. 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.

3. 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 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.
4. A program takes longer to load if it uses large code than if it uses small code. Instructions in a large code program that reference a code segment location must be adjusted when the program is loaded, since such instructions must contain the absolute address of the location and since this is not known until the program is loaded.

Instructions in a small code program that reference code segment locations need not be adjusted, since they are always independent of the location at which the code segment is loaded: If the location is within 32K of the referencing instruction, the instruction is pc-relative; and if it is outside this range, the instruction is a position-independent jump to a jump table entry.

When a small code program that contains a jump table is loaded, its jump table entries must be adjusted, since these are jump instructions to code segment locations, where each instruction must contain the absolute address of the destination address. However, it should take less time to adjust the jump table for a small code program than to adjust the code segment of a large code version of the same program, since for any destination of a jump or call instruction a small code version of a program will have at most one jump table entry needing adjustment, whereas a large code version of the program may have many jump or call instructions to the same location that need to be adjusted.

5. 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.

6. 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 is not specified, the module will use small code.
- md** Module uses large data. If this option is not specified, the module uses small data.

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

cc prog	small code, small data
cc -mc prog	large code, small data
cc -md prog	small code, large data
cc -mcd prog	large code, large data

Libraries

The modules in the Aztec C libraries use the small code, small data memory model. Most libraries have large code and large data equivalents, e.g., **cl.lib** is the large data version of **c.lib** and may be linked in with **-lcl** instead of **-lc**.

Unless you have extraordinary requirements, you will probably never need to use **-mc** on the Amiga. If one of your functions calls another function in the same segment that is more than 64K bytes away, the linker automatically forces the target routine into the application's jump table and directs the call through the jump table. See the *Amiga ROM Kernel Reference Manual* for a description of the jump table. Function calls across segment boundaries always go through the jump table.

Furthermore, your first code segment may be of arbitrary size, again without the use of **-mc** or any other special action on your part. The application runtime startup code (**crt0.o**) takes care of fixing up the jump table, using information supplied by the linker.

When you compile a module with **-md**, all references in that module to global and static data are set up as 4-byte absolute values. The use of **-md** is never necessary until the size of the initialized and uninitialized data segments (see the output from the linker) exceed 64K bytes. Large-data references, Data-to-data references or code-to-data references, are relocated by the DOS loader.

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 creates 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, address register **a4**, which the small data modules use to access data, points 32K from the beginning of this data segment.

Here is the caveat: data that the small data modules attempt to access must be within 32K bytes of the location pointed at by **a4**. The linker detects data accesses by small data modules for which this condition is not satisfied, and issues a message. If you get this message, try reordering the way in which the linker encounters them. If that does not solve the problem, you will have to recompile the small data modules, to make them use large data.

Code Segmentation

By default, the linker creates a program that has a single code segment. As you have seen, there is no limit on the size of this segment, regardless of the memory model used by the program.

However, there is a limit on the amount of memory in a machine. Also, the larger a program's code segment, the worse its performance. If the program uses small code, its execution speed degrades as the program gets larger, because more transfers of control will have to be done indirectly, via the jump table. And if it uses large code, it will be larger and slower than a small code version of the same program.

The Aztec linker allows you to divide a program's code into multiple segments. When the program is running, code segments are brought into memory as needed. When a code segment is no longer needed in memory, the memory it occupied can be released and reused, either for another code segment or for some other purpose.

One advantage of partitioning a large program into code segments is that it allows a program to be larger than available memory. Another advantage is that it allows the code to use small code memory model without the degradation caused by indirect transfers of control via the jump table.

There is some loss of performance in a segmented program, since the segments may be loaded into memory from disk more than once to be executed. With careful partitioning of the program, the loss can be minimized.

Compiler Options

Option Philosophy

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 to 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 Chip or processor control
- f Floating point control
- h pre-compiled header file control
- m Memory model control
- p Parser control
- q Output control
- 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. -p0t, for instance, turns off trigraphs and -pt turns them on.

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 produces the desired result. -p0ta would not produce the intended result. Since the a option is scanned after the 0t option, the 0t option is cancelled.

CCOPTS Environment Variable

You can override the default settings of the compiler by using the environment variable CCOPTS.

If you want to use 16 bit ints as the default, for example, you could say

```
set CCOPTS=-ps
```

which would cause the compiler to use 16 bit ints instead of the standard default of 32 bits.

The CCOPTS specification should have no embedded blanks including both sides of the equal (=) sign. If blanks are used as separators the whole specification must be enclosed in quotes. Thus:

```
set CCOPTS=-ps
```

or

```
set "CCOPTS = -ps"
```

but not

```
set CCOPTS = -ps
```

Options passed directly to the compiler override the CCOPTS environment variable. If the CCOPTS environment variable was set to **-ps** and you specified **-p0s** as a direct option to the compiler, then the CCOPTS **-ps** option would be overridden.

If you wish to specify more than one option with the CCOPTS environment variable, then each option group must be separated by a blank and the whole string enclosed in quotes. For example,

```
set "CCOPTS=-ps -mcd -wo"
```

would set the **-ps**, **-mc**, **-md**, and **-wo** options. Note that CCOPTS must be specified in upper case.

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

Option List

- 3** Interpret options that follow according to version 3.6 rules
- 5** Interpret options that follow according to version 5.0 rules
- a** Causes the compiler not to start the assembler after it has compiled a program. Only an assembly file is created.
- at** Same as -a, but also imbeds C source statements into the assembly code.
- bd** Enables stack depth checking.
- bs** Produces sdb debugging information
- c2** Compiler will create 68020 code.
- d** Defines a symbol for the preprocessor.
- fa** Generates code for Amiga IEEE floating point.
- ff** Generates code for Motorola Fast Floating point.
- fm** Generates code for Manx IEEE. Defaults to on.
- f8** Generates code for 68881 Floating Point format.
- hi** Reads pre-compiled header files into its symbol table.
- ho** Writes compiler symbol table. Used for making pre-compiled header files.
- i** Specifies path for include files.
- k** Compile code according to the K&R (UNIX version 7) standard
- ma** Forces alignment of short and long data items. Defaults to on.

- mb Generates the public .begin statement. Defaults to on; turn off for drivers.
- mc Generates code that uses large code memory model.
- md Generates code that uses large data memory model.
- me Aligns strings on word boundaries. This forces all strings to start on an even address.
- mm Puts data in code segment.
- ms Puts static strings in data segment.
- o Specifies name to be used for output file.
- pa Turns on ANSI preprocessor and trigraphs. Turns off 'cdemou' suboptions for -p.
- pb make bitfield unsigned by default
- pc Allows extra characters after a #endif or #else. Defaults to on.
- pe Causes enums to occupy only the amount of space needed. Defaults to off.
- pl Defines integers to be 32 bits instead of 16 bits. Defaults to on.
- po Uses old (3.6) preprocessor.
- pp make characters unsigned by default
- ps Defines integers to be 16 bits instead of 32 bits.
- pt Looks for trigraphs in the input stream. Defaults to off.
- pu Uses unsigned preserving rules.

- qa** Causes generated prototypes to use __PARMS() syntax.
- qf** Enables QuikFix.
- qp** Generates prototypes for all non-static functions defined within the current file.
- qq** Disables the startup and error messages from being displayed
- qs** Generates prototypes for all static functions defined within the current file.
- qv** Generates verbose information on memory usage.
- r4** Uses a4 for register variables.
- sa** Enables two pass assembly for branch squeezing and other optimizations.
- sb** Enables use of built-in in-line functions for the string operations `strcpy()`, `strcmp()`, and `strlen()`.
- sf** Generates an optimized `for(;;)` loop that places the test at the bottom of the loop.
- sm** Defines 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, does not generate the `LINK` and `UNLK` instructions.
- so** A shorthand way to specify the suboption 'afmnprs' for -s.
- sp** Delays the popping of arguments until necessary.
- sr** Allocates registers based on weighted usage counts.

- ss** Finds duplicate strings and replaces them with a pointer to the first occurrence.
- su** Allocates registers based on weighted usage counts, but allocates to user specified variables first.
- wa** Complains on arguments which do not match the prototype specification.
- wd** Generates warnings for old style K&R definitions.
- we** Quits on warnings. Treats warnings as errors.
- wl** Shorthand for **-waru** and stands for lint.
- wn** Do not generate warnings on direct pointer to pointer conversions
- wo** Causes pointer/int conflicts to generate warnings rather than errors.
- wp** Generates a warning if a function is called without a prototype being defined for a function.
- wq** Prints warnings to the file **aztecC.err**.
- wr** Warns if function return type does not match declared type.
- ws** Ignores all warnings.
- wu** Warns about unused local variables.
- ww** Allows compiler to continue beyond 5 errors.

Option Descriptions

Option -3

The option(s) following this option will be interpreted according to version 3.6a syntax. The compiler option specification scheme for version 5 was reorganized to make it more coherent and to allow for future requirements. The **-3** option is provided to reduce the impact of conversion on version 3.6a users. The document, *The Transition from Aztec C version 3 To Aztec C version 5 On The Commodore Amiga*, should be consulted. The following is a simple example of the use of this option,

```
cc -3 +x3 +c +d prog.c
```

There are several fine points to using the **-3** option.

- ❑ The default integer size in version 5 is 32 bits. The default in the version 3 compiler and earlier versions was 16 bits. The **-3** option sets the default integer size to 16 bits to maintain compatibility. If 32 bit integers are required, then specify the **+p** or **+l** options following the **-3** option.
- ❑ If the **-3** option is used, options using version 5 syntax can be specified before the **-3** option or following the **-5** option.
- ❑ The **-k** option is a special case and can be used anywhere in the option specification list.
- ❑ The version 3.6a option **+x1** is no longer supported.
- ❑ The version 3.6a table size options, (**-e**, **-z**, and the like) are accepted but have no effect. The version 5 compiler allocates its internal memory space dynamically.
- ❑ The version 3 **+e** option is accepted but has no effect. This option is no longer necessary since version 5 always preserves the registers that were preserved by this option.

Setting the **CCOPTS** environment variable to **-3k** is a simple effective way to delay changing makefiles or other command files that invoke the Aztec C compiler using the old style option syntax. The command line looks as follows,

```
set CCOPTS=-3k
```

The **-3** option is not a global version 3.6a compatibility switch. For a discussion of library routines, library names, ANSI syntax changes and other version 3.6a to version 5 differences it is strongly advised that you consult the transition document cited earlier.

Option **-5**

The option(s) following this option will be interpreted according to version 5.0 syntax. This option is used with the **-3** option to allow the specification of options using version 5.0 syntax after the specification of options using version 3.6a syntax. For example:

```
cc -3 +p -D GRAPHICS -e1000 -5 -sn prog.c
```

Option **-c2**

Directly generates 68020 instructions that will take advantage of the 68020 and 68030 chips. These instructions will not run on a 68000 or 68010.

Option **-d macro**

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

```
cc -dmacro[=text] filename
```

For example,

```
cc -d MAXLEN=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:

```
cc -d DEBUG 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. For example,

```
#ifdef AMIGA
amigainit();
#else
ibminit();
#endif
```

Option -fa

Use this option when linking with the **ma.lib** library. Doubles and long doubles are represented as 64 bit IEEE format and floats use 32 bit IEEE. For support for long doubles, use the **-fm** option listed below.

Option -ff

Use this option when linking with the fast floating point **mf.lib** library. Float and double are represented as 32 bit values in this library and long doubles as 64 bit values.

Option -fm

This is the default math option. Use it when linking with the Manx IEEE **m.lib** library. It supports the long double type as 96 bit format, double as 64 bits, and float as 32 bit.

Option -f8

Use this option when linking with the **m8.lib** library. It generates code to directly use the 68881 floating point chip.

Option -hi *filename*

This option specifies the name of an input file containing pre-compiled header file information. The input file is created using the **-ho** option.

Separating space between the identifier **-hi** and the filename is optional.

Option -ho *filename*

This option specifies the name of an output file for pre-compiled header information. This information is later used with the **-hi** option to compile programs that use the same header files. The use of pre-compiled header files can significantly shorten compile times.

If you are including large header files, like **functions.h**, in your programs you should use pre-compiled header files.

Separating space between the identifier **-ho** and the filename is optional.

Only one pre-compiled header file may be used per compilation, but it may contain information from numerous header files. For example, if the C source files for a program included the header files **stdio.h**, **fcntl.h**, **math.h**, and **functions.h**, you could speed up the compilation by precompiling these files. To do that, create a file named **x.c** that consists of the following lines:

```
#include <stdio.h>
#include <fcntl.h>
#include <functions.h>
#include <math.h>
```

Create the pre-compiled header file **x.dmp** using the following command line:

```
cc -ho x.dmp x.c
```

Then, when compiling a source file that includes any of these header files, specify **-hi x.dmp**. If the source file was named **prog.c** then the command line would look as follows:

```
cc -hi x.dmp prog.c
```

The use of pre-compiled header files does not require any changes to your source files. The **#include** statements remain without change. The only difference between programs compiled with and without pre-compiled headers should be the **-ho** specification.

The list of header files referenced in your source programs and those contained in the pre-compiled header file do not have to perfectly match. The source program can reference header files that are not pre-compiled and the pre-compiled header file can contain headers that are not referenced in the source program. The compiler will extract header information from the pre-compiled header file and then generate any additional header information that is needed.

The use of pre-compiled header files may seem a bit complicated at first, but for programs that include large header (.h) files the time savings is worth the effort of learning how to use them.

Option -i path

Option **-i** causes the compiler to search in a specified area for files included in the source code.

The name of the area follows the **-i** with an optional separating space. For example, the following defines directory **source/inc** on volume **sys:** search area:

```
-i sys:source/inc
```

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 3.6a or some other K&R conforming compiler. The **-k** option is equivalent to compiling with the options **-pou0a -wo**.

Option **-mb**

By default the compiler generates:

```
public .begin
```

The compiler generates the reference to **.begin** to force loading of the program startup code that eventually calls **_main()** from the library. If you are writing a driver or other piece of stand-alone code, then you do not want the standard program startup. So, to avoid getting the standard startup, you can either define your own **.begin** symbol, or compile all files with **-m0b** to prevent the **.begin** statement from being generated. All libraries are compiled with **-m0b**.

Option **-mc**

Causes the compiler to define the name **_LARGE_CODE**, forcing the assembler to generate 32 bit absolute code references rather than **a4** relative jump table references.

Option **-md**

Causes the compiler to define the name **_LARGE_DATA**, forcing the assembler to generate 32 bit absolute data references rather than **a4** relative references. This memory model is used in some of the header files to switch between an external definition of some hardware addresses and a macro definition with the address hard-coded in.

Option **-mm**

This suppresses all **dsegs** from being generated by the compiler, which causes code and data to be intermixed. This might be useful for some ROM applications and has been included here for compatibility with Greenhills C.

Option -o filename

This option is used to override the default output filename. The separating space between the identifier **-o** and filename is optional. A detailed discussion of compiler output files appears earlier in this chapter.

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 should compile 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 -pe

By default the size of **enums** is the same as that of an **int**. This option places them 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=5, Germany}
```

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

Option -pl

Integers are represented as 32 bit format. This is the default setting and would be used in conjunction with the **c.lib** library.

Option -ps

Integers are represented as 16 bit format. With this setting, care needs to be taken in calling Amiga library routines which expect 32 bit integers or passing integer values to pointers which expect 32 bit values. Use the **c16.lib** library when using this setting.

Option -pt

Trigraphs are an attempt by the ANSI committee 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 parser, but it must be supported for ANSI compliance. The **-pa** turns on trigraph checking, but this option is included to turn off trigraphs while retaining the rest of the ANSI compliance.

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 _ANSI_C
#define _PARMS(x) x
#else
#define _PARMS(x) ()
#endif
```

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

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

Otherwise if `_ANSI_C` 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 -qf

The `-qf` option enables QuikFix which is described in the *Aztec C User Guide*.

Option -qp

The `-qp` and `-qs` options control whether static functions should have a prototype generated or not. Both options may be selected together to get all the functions. However, generally, all the global definitions are combined into one header file included by all files, while static function prototypes are placed in each individual file. See also `-qa`

Option -qq

The `-qq` option disables the startup and error count messages from being displayed on the screen.

Option -qs

See the `-qp` option.

Option -qv

The `-qv` or verbose option displays information on memory usage to the screen.

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. 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 '}' on which to hang it. 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**

Normally, when a function is called, it creates a stack frame using the **LINK** instruction to save register **a5**, point it at the old **a5**, and decrement the stack pointer by the amount of local storage needed. Then all references to local variables and arguments are made through **a5**. If the **-sn** option is selected, and there is no local storage needed by the function, then the **LINK** and accompanying **UNLK** instructions are not needed and are eliminated. Arguments are accessed using the stack pointer directly.

(Note: This confuses **sdb** and causes it to run more slowly.)

Option **-sp**

This option saves some small amount of execution time when several functions are called one after another. This is accomplished by delaying stack operations until necessary and thereby eliminating unnecessary stack operations. The stack is corrected whenever it is necessary or if the number of bytes delayed exceeds 50 bytes.

Option **-sr**

-sr causes the compiler to pick which local variables should be assigned to register variables. 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 strings 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 "ANSI-izing" your programs.

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 -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 -wo

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

Option -wp

This 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.

Option -wq

By default the compiler displays error messages on the screen. This option sends them to a file named **AztecC.err**. This file is used by the QuikFix facility.

Option -wr

Normally, a function is declared without a type, which implies that it returns an **int**. However, if there are any return statements, explicit or implicit, that do not return a value or that return a value whose type disagrees with the type of the function, then a warning is generated for the return statements. 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

Since the compiler has the entire tree of the function in memory, it can detect that a variable has been declared but has not been used. This option directs the compiler to check for such variables and generate 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.

Table Manipulation Options

The compiler contains several memory-resident tables in which to store information about a program it is compiling.

Note to previous users: Prior to Version 5.0, these tables were static in size with command line options being used to alter the default sizes. The compiler now dynamically allocates and frees space for all of its internal data and tables. Therefore all previous options which dealt with table sizes have been eliminated.

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 cursor highlighting the approximate point where the error occurred.

The compiler then displays a line containing the following information:

1. the name of the source file containing the line
2. the number of the line within the file
3. an error code
4. a message describing the error
5. the 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:

```
cc > prog.err prog
```

When the compiler sends error messages to the screen, it pauses after five error messages are displayed, and asks if you want it to continue. If you type **Y**, followed by a <CR>, the compiler continues.

If you type anything else, followed by a <CR>, the compiler stops.

When the compiler sends error messages to a device or file other than the screen, it processes the entire file without giving the operator the opportunity to abort the compilation, even if it detects errors.

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.

Writing Programs for the Amiga

#pragma for Amiga Exec Calls

Normally, when a function is called in a program, the code for that function is physically made a part of the program by the linker. This is true for all the Manx supplied functions that are in the library we provide. On the Amiga, there are a number of additional libraries which are used to interface the program with the operating system and the graphics environment.

Many of these libraries are located in ROM, although some are loaded from disk into RAM. Since these routines cannot be loaded into the program, and since the physical address of each routine can be different from one version of the ROM to another, a dynamic system of calling ROM library routines exists on the Amiga.

The structure and use of resident libraries is discussed in the Amiga Rom Kernal Manuals. In brief, each library has a pointer to a library structure. Preceding this structure is a jump table with entry points for each function in the library. When a library is opened, the pointer to the library structure is returned. When a call is made to the library, parameters are placed in the appropriate registers, the library pointer is placed in register **a6** and the appropriate negative offset is called.

In versions of Aztec C prior to 5.0, when a ROM library routine was called, the arguments were pushed on the stack and a small assembly language stub was added to the program which pulled the arguments off the stack into the appropriate registers, loaded the library base pointer into **a6**, and called the routine. This resulted in a number of small glue routines which added overhead both in terms of size and speed.

The functions pointed to by the library vector would also be an assembly language stub that would save registers, set up appropriate registers, push the register parameters onto the stack, and then call the function which would handle the request.

Aztec C now also allows the compiler to directly generate the call to the ROM library by generating code to load arguments directly into registers, load the beginning address into **a6** and call the routine. This function is implemented using a **PRAGMA**.

Pragmas are the ANSI committee's way of allowing extensions to the C language in a controlled manner. Essentially, any line beginning with **#pragma identifier** is parsed by the compiler to see if *identifier* matches any that the compiler knows about. If not, the line is ignored.

Calling Resident Libraries

Making a call to a resident library is very simple with version 5.0 of Aztec C through the use of an **amicall #pragma** statement. (#pragma statements

are the ANSI supported way of doing magic things with the compiler.) An **amicall** statement takes the form:

```
#pragma amicall (LibBase, offset, func(reg1, reg2, ..., regn))
```

In this statement, the *LibBase* parameter is the name of the variable which contains a pointer to the library as returned by **OpenLibrary()**. *offset* is the positive offset value for this function. *func* is the actual name of the function being called. *reg1* through *regn* are the registers in which the parameters are passed. For example, the Exec call **AllocMem()** has a pragma defined as follows:

```
#pragma amicall(SysBase, 0xc6, AllocMem(d0,d1))
```

which declares that *SysBase* is the variable containing the pointer to Exec's library base structure. The routine to be called is at offset - (0xc6) from the *SysBase* pointer. The name of the routine is **AllocMem()** and the two arguments are to be passed in registers **d0** and **d1** respectively.

Now, when the compiler sees a statement like:

```
ptr=AllocMem(sizeof(struct Node), \
MEMF_PUBLIC|MEMF_CLEAR);
```

some form of the following code is generated:

```
move.l      a6, -(sp)
move.l      #14, d0
move.l      #$1000, d1
move.l      _SysBase, a6
jsr         -$c6(a6)
move.l      (sp)+, a6
move.l      d0, _ptr
```

Thus, no assembly language glue routine is required, since the compiler handles it all automatically.

For the resident libraries that are a basic part of the Amiga computer, ANSI style prototypes and **amicall** pragmas are defined in the file **functions.h** which is found in the standard Aztec C include directory. Thus, any file which contains a reference to an Amiga library call should:

```
#include <functions.h>
```

Since there are over 500 Amiga functions, pre-compiling `functions.h` is greatly recommended (see the compiler `-hi` and `-ho` options).

So when you are creating your own resident library, the only thing you need to do to allow Aztec C programs to make calls to your library is to define the appropriate `#pragma amicall` statements in a header file and the compiler will do the rest of the work.

Creating Resident Libraries

Creating resident libraries is now very simple.

The distribution disks contain two files that can be modified to create your own resident library. The files are `libstart.asm` and `libsup.c`. These files along with file `lib.c` are a simple example of the components of a resident library.

The `libstart.asm` file contains the assembly language rom tag structure and the assembly language initialization required for Aztec C's small model. To use this routine in creating your own resident libraries, the only thing that needs to be modified is the version number and priority. To modify them reference the macros `MYVERSION` and `MYPRI` at the beginning of the file.

The `libsup.c` file contains data structures and four routines that do initialization and support. The first data structure is `libfunctab`. The first three entries in this structure are mandatory resident library entries and point to routines found in `libsup.c`. The fourth entry is reserved. The following entries are pointers to the user defined library routines. The last entry is the end of table code which is a negative one (-1). The sample routines `func1` and `func2` that are coded in this sample should, of course, be replaced with names of your own routines.

The `myInitTab` structure does not need to be changed. It points to the `libfunctab` structure and to the startup code in `libstart.asm`.

Next you will find data definitions that give the revision level, the name of the library, and an 'id' string. The library name should be set to the name of the library you are creating. The other names, `myname` and `myid`, should

not be changed unless the **libstart.asm** file is modified to mirror the changes made.

The first routine, **_main**, is called by the startup code when the library is first loaded into memory and only then. Once the library is opened, additional calls to **OpenLibrary** will call **myOpen** which will increment the library use count by one. Each time **myClose** is called, the library use count is decremented by one.

The **myExpunge** routine is called when the system requires more memory. If the library use count is zero, the resident library is removed and the associated memory is freed. If the use count is not zero, then a flag is set so that the library will be automatically removed when the open count drops to zero. These routines do not have to be changed although they can be changed.

The example has two routines **Func1** and **Func2** in the file **mylib.c**. The **Func1** routine stores the number passed to it and the **Func2** routine adds the number passed to it to the number saved by the **Func1** routine. Note that the only thing special about these routines is the declaration of an uninitialized variable matching the name of the library base. The compiler will automatically assign this variable to the register **a6** which contains a pointer to the library base when this routine is called.

The parameters to the function are actually in registers when the function is called. However, the compiler will generate code that will correctly deal with the the parameters because of the **amicall** pragma for each function which is defined in the **mylib.h** file.

The file, **test.c**, contains an example that uses the resident library just described. Note that the file **mylib.h** is included in **test.c** and it specifies which register to use for each parameter.

#pragma for Register Function Calls

It is possible to use registers to pass arguments to regular functions. To do this use the **regcall** pragma which is defined similar to the **amicall** pragma.

```
#pragma regcall([return=]funcname (arg1, arg2,...,argn))
```

This is interpreted in the same manner. This pragma will affect both the call to the function as well as the function itself which will look in the specified registers for arguments.

Miscellaneous Notes on Resident Libraries

- Any function referenced in an **amicall** or **regcall** pragma must have been previously declared (preferably with a prototype of the arguments). For example:

```
#pragma amicall(mylibBase, 0x1e, func1(d0))
```

by itself will generate an error, while:

```
void func1(long value);
#pragma amicall(mylibBase, 0x1e, func1(d0))
```

is correct.

- The pragmas, **libcall** and **syscall**, are parsed and handled just like **amicall**.
- If the compiler is invoked with small code or data, each function with an **amicall** pragma saves **a4** on entry and loads it with a pointer to the libraries data. On exit **a4** is restored. If invoked with large code and large data, **a4** is not affected.
- The file **mylib.fd** is included on the distribution disk as an example of a program definition file for the MAPFD program. The file specifies parameter passing conventions and function names. The MAPFD program then produces a set of #pragmas and assembly language glue routines. You can follow this as a guideline for process your own libraries with MAPFD. The **.fd** file you create should be saved for possible use with an MAPFD program for BASIC or some other language.

Pointer Considerations

Pointers are 32 bits wide in Aztec C compiled programs, whereas ints may be 16 or 32 bits wide. Because of this difference, a program whose ints are 16 bits wide will not work if the program assumes that pointers and ints are the same size, and that pointers and ints are treated the same. Since by default an int is 32 bits wide, this is only a problem when compiling with the -ps option.

It is easy for a C program to accidentally or purposely make these assumptions, since in C the type of an undeclared function or function argument is assumed to be int. A program making these assumptions runs on machines for which the assumptions are true, of course. But when you use Aztec C to recompile the program, it will malfunction.

A program that uses pointers should obey the following rules:

- If a function returns a pointer, explicitly declare it, both in the function itself and in any module that calls the function.
- If a function argument is a pointer, explicitly declare it in the function itself, and be careful not to accidentally pass an int as the argument.

The following paragraphs discuss these rules in detail.

Declare Functions That Return a Pointer

The following code demonstrates the importance of declaring a function that returns a pointer, both in the function itself and in a function that calls it.

For this example, assume that the function g must be passed a pointer.

```
char *f();  
...  
g(f());
```

The compiled code passes the 32-bit pointer returned by the function f to the function g. If f was not declared as returning a pointer, the compiled

code, when using the **-ps** option, assumes that a 16-bit int was returned by **f**, and hence passes only part of the pointer returned by **f** to **g**.

Declare Function Arguments That Are Pointers

The following code demonstrates the importance of declaring that a function argument is a pointer in the function itself.

For this example, assume again that the function **g** must be passed a pointer.

```
f(y)
char *y;
{
...
    g(y);
...
}
```

The compiled code takes the 32-bit pointer **y**, that is passed to the function **f**, and passes it to the function **g**. If the declaration **char *y** is omitted, the compiler, when using the **-ps** option, assumes that **y** is a 16-bit int, and hence generates code that passes only part of the pointer to **g**.

Pointer Variables And Constants As Arguments On Calls

Code that passes integer constants or variables as function arguments where pointers are expected may not work with Aztec C.

One common problem is attempting to pass the constant **int 0** as a null pointer. For example, if **g** is again a function that is passed a pointer, the following code demonstrates one way to correctly pass a null pointer to **g**:

```
g((void *) 0);
```

(The code does not have to cast 0 to be a pointer to a **void**; casting it to be a pointer to any other type of object also would work.) If, instead, the code says:

```
g(0);
```

with the **-ps** option, a 16-bit null value is passed, which would be wrong.

Other Considerations

Internal Storage Of Numeric Data

Programs written for processors that store data items in least significant to most significant order may need to be changed. The MC68000 processor stores data in most significant to least significant order. This is true of both non floating point and floating point data.

The following short program illustrates a program that runs differently depending on the manner in which `int` data items are stored.

```
cput(c)
int c;
{
    write(1, &c, 1);
}
```

Problems can also occur reading data created in another environment where the data is stored in the reverse order.

The MC68000 requires that any memory access must be aligned on an even address unless it is a single byte access. The Aztec C compiler aligns nonbyte data items on even boundaries to avoid memory faults. Code that accesses nonbyte data through pointers that specify an odd memory address causes a system crash.

Converting Data

Programs reading data written in another C environment that does not force alignment will probably not produce correct results. Programs writing data that will be accessed in an environment that does not force alignment will likewise probably fail.

Some conversion will probably be needed to insert slack bytes to assure even alignment when importing data created for an unaligned environment, and to remove slack bytes when exporting data for an unaligned environment.

Most of the common 8-bit microprocessors store numeric data in an order that is the reverse of the MC68000. The 808x 16-bit processors and the PDP-11 also store numeric items this way.

Additional Features

Line Continuation

If the compiler finds a source line whose last character is the backslash character \, it considers the following line to be part of the current line, without the backslash. For example, the following statements define a character array containing the string **abcdef**:

```
char arr[]="ab\
cd\
ef";
```

String Concatenation

Character string literals that are adjacent tokens are concatenated into a single character string literal. For example, the following statements define a character array containing the string **abcdef**.

```
char arr[] = "ab" "cd" "ef";
```

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
#endif
/* 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).

For a discussion of register usage by assembly language programs, see the Programmer's Information section of the **Assembler** chapter.

For a discussion of accessing macro definitions, arguments, and automatic and static variables from within a **#asm** block, see the **Technical Information** chapter.

LANGUAGE SPECIFICATION

3

Chapter 3 - Language Specification

Compatibility Issues

The Aztec C68K compiler for the Amiga is entirely compatible with the proposed draft ANSI standard for C. This means that porting programs which conform to the ANSI standard to the Aztec C68K compiler should be a very straightforward process, with few if any modifications to the code.

In addition to ANSI compatibility, Aztec C also has a number of library functions and compiler switches which provide a fair level of UNIX v7 C, UNIX System 3 C, UNIX System V C, and XENIX C compatibility.

Compatibility with Other Aztec Compilers

Within major release numbers, code may be easily moved from one implementation of Aztec C to another. For example, moving code from v3.6a of the Manx Amiga compiler to v3.6c of the Manx Macintosh compiler is fairly simple (except for the differences in machine-specific library functions). Where release numbers differ (i.e., 3.6c and 5.0a), code is generally upward compatible, but some changes may be needed to move code down to a lower release.

Note that compatibility between machines exists only with respect to source compilation and use of standard library functions. Machine-specific functions, such as Amiga Workbench calls, have no direct equivalents on other machines such as PCs or Macintoshes. The only library functions

which are portable across Aztec C implementations are those marked "Aztec" or "ANSI" in the library function chapter.

Keywords

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

Operators

[] () . -> ++ -- & * + - ~ ! / % << >> < > <= >= == != != && || ? : *= /= %= += -= <<= >>= &= ^= |= , # ##

Pre-Processing Directives

#if	#ifdef	#ifndef	#elif
#else	#endif	#include	#define
#undef	#line	#error	#pragma
#	#asm	#endasm	

Type Information

The types supported by Aztec C68K and their sizes are listed in the following table.

Base Type Sizes

<i>TYPE</i>	<i>SIZE (in bits)</i>
char	8
short int	16
int	16 or (32)
long	32
float	32
double	(32) or 64
long double	32, 64, or (96)
function pointer	32
data pointer	32

In cases where there is more than one possible size, the default is delineated by (). Signed types are the same size as their unsigned counterparts.

Integers and Long Integers

Integer and long values are stored internally from most significant byte to least significant byte (i.e., the normal Motorola 68000 format). This can cause problems with programs ported from architectures which use other conventions for 16-and 32-bit storage, such as the 8086. For example, a program designed for the 8086 which accessed 16-bit values one byte at a time would not work properly. Programs that access ints and longs in the normal manner should function normally.

Integers and long integers are also always aligned on an even-address boundary. This alignment is required by the 68000 CPU when it has to access a 16 or 32-bit value. Programs which attempt to access byte-sized data in an integer manner will most likely crash the program. In this example

```
char *byte_nums = {0, 1, 2, 3, 4};  
int *word_nums, sum;  
  
word_nums = (int *)byte_nums+1;  
sum = *word_nums + *(word_nums + 1);
```

the last line will cause a 68000 exception number 3 (address error).

Characters

The `char` type is always considered as signed by Aztec C. An unsigned character must always be explicitly declared as `unsigned char`. This may cause portability problems when porting code from other compilers/platforms. For example, the following code will not work as it would on a machine which considered `char`s unsigned:

```
char a = 255;

if (a > 30)
    printf ("a is bigger than 30\n");
else
    printf ("a is smaller than 30\n");
```

Although `a` appears to be set to 255, it is actually considered to contain -1 (255 translates to 1111111 binary, which is the two's-compliment representation of -1). Therefore, the message `a is smaller than 30` will be printed. On a machine with unsigned characters, the message `a is bigger than 30` would be printed.

Structures

Due to alignment requirements of integer and long values, padding bytes, also known as "holes", may exist within a structure. For example:

```
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.

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

```
sizeof (a_struct)
```

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

Bitfields

Bitfields are fully supported by Aztec C68K. Bitfields are numbered as per normal 68000 convention. For example:

```
struct {
    unsigned int bf1 :1;
    unsigned int bf2 :1;
    unsigned 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 bit field 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.

Enum Constants

By default, **enum** constants are always represented as an **int**. However, a compiler switch will cause **enums** to be sized to the smallest possible type (**char**, **short**, **int**, or **long**).

Const and Volatile

Type checking for the **const** keyword is fully supported by Aztec C68K. This means that the compiler will flag the following code as an error:

```
const char c_char = 10;
c_char = 3;
```

The **volatile** keyword is also semantically as well as syntactically supported. The compiler will turn off all optimization for any variables of type **volatile**. More specifically, the compiler will never perform register optimizations for **volatile** variables. The value of the variable will always be fetched from the variable's location in main memory.

General ANSI-Compatible Features

Aztec C68k fully supports the ANSI standard. The following describes some of the more important ANSI features.

Function Prototyping and ANSI Function-Definition

ANSI function prototypes and ANSI-style function definitions allow the Aztec C compiler to identify incorrect specifications of function arguments. A function prototype is a function declaration which informs the compiler of the number and type of arguments that the function takes, and the function's return type. With this information, the compiler performs extensive type-checking on function arguments and identifies common errors, such as accidentally passing a character pointer instead of a character to a function like `putc()`.

There are three C constructs involved in function prototypes: function declarations, function definitions, and function calls. We will first consider the use of these components as described by Brian Kernighan and Dennis Ritchie in the book *The C Programming Language* (from here on referred to as "K&R"), which defines the version of C used by Aztec C prior to version 5.0:

FUNCTION DECLARATION - Function declarations in K&R only identify the return value of a function. This type of declaration will be referred to as an **OLD-STYLE** declaration. The following is an example

```
① double sin();  
②  
③ int positive_sin(d);  
④ double d;  
⑤ {  
⑥     d = sin (d);  
⑦     return (d > 0.0);  
⑧ }
```

Line 1 is a DECLARATION of the function `sin()`, and states that `sin()` returns a double-precision floating point number. Without this declaration, the compiler assumes that `sin()` returns an `int` instead of a `double`. This assumption would cause `d` to contain a random and

incorrect result. It is important to note that a function's return-type is the *only* information conveyed in old-style definitions.

FUNCTION DEFINITION - The function definition is the one and only place where a function is defined. The definition specifies the function's return type, arguments, and the actual code comprising the function (the body of the function). Using the above example, lines 3-8 comprise the full definition of the function **positive_sin()**. Lines 3 and 4 define the function name, return type, and arguments, and lines 5-8 comprise the function body.

In old-style function definitions, an important rule to remember is that arguments of type **short** or **char** are *always* converted to type **int**. Arguments of type **float** are *always* converted to **double**.

If you pass a function a variable of type **float**, for example, the compiler will automatically convert it to type **double** before passing it to the function (this is called **PROMOTION**). Thus, it's impossible to pass variables of types **char**, **short**, and **float** directly to a function when old-style definitions are used.

FUNCTION CALL - Line 6 in the above example shows a call to the function **sin()**, with the double-precision variable **d** as an argument, and **d** also receiving the return value of **sin()**. By looking at the function declaration of **sin()** on line 1, we know that **sin()** does indeed return a double value. If we tried to equate a variable of type **int** to **sin()**, the compiler would issue a warning message that the variable did not match the return type.

However, it is not apparent from the function declaration exactly how many arguments **sin()** takes, nor what their types may be. In fact, the compiler does not know the arguments received by **sin()** either, so *any* variable could be passed to **sin()** without any compiler warnings. Thus, we must rely on memory or reference manuals to insure that the correct arguments are passed to **sin()**.

ANSI Standard

The ANSI standard has extended the syntax of both the declaration and definition of functions to allow the number and types of arguments to be

specified in function declarations. For example, the above example could be re-coded as:

```
double sin(double x);

int positive_sin (d)
double d;
{
    d = sin (d);
    return (d > 0.0);
}
```

Notice that the first line now contains **double x** within the parenthesis. This informs the compiler that the **sin()** function takes an argument of type **double** (the **x** is there simply for readability; the compiler ignores it). The compiler will do appropriate error checking. Thus, if we tried to say:

```
d = sin (4);
```

the compiler will issue a warning message that the type of **4 (int)** does not match the function's required argument type (**double**) and would then convert the **int** into a **double**.

Look at this a little closer, using another example:

```
void other (float size, long time, short stk);

void test (char c, float f, long l)
{
    other (c, l, l);
}
```

The first line indicates that **other** has no return value (**void**), and takes a **float**, **long**, and **short** argument (in that order). The declaration of function arguments has some important ramifications:

- Each argument will be converted, as if by assignment, to the corresponding type specified in the declaration. Thus, **c** will be converted from a **char** to a **float**, the integer constant **l** will be converted to a **long**, and **l** will be truncated from a **long** into a **short**.

- In addition, the compiler will warn that these three items are being coerced to fit the function declaration (these type of warnings can be turned off with compiler switches). The compiler will also check to insure that the correct number of arguments are being passed and will terminate with an error if too few or too many arguments are passed to the function.
- Default promotions are NOT done on **char**, **short**, and **float** values. Thus the **float** parameter in the above example will always be passed as a **float**, and not as a **double**. In essence, the prototype function declaration defines the default conversions to be done.

It is also important to note that when ANSI prototype-declarations are used, a new style of function definition is also used. Rather than declaring the names of the arguments in parenthesis, then declaring the argument types below that, the ANSI style declares the argument types and names together within the parenthesis (as is done on the third line). As a general rule, when function prototypes are used for a function, you should use the ANSI-definition style. In the absence of prototypes, you may use either style.

To summarize:

- If a function definition uses the new ANSI-style, arguments are treated exactly as specified in the argument list, regardless of whether a new or old-style declaration exists.
- If a function definition uses the old K&R style, arguments are treated as though they have been promoted.
- If there is an old-style or missing declaration, no checking on argument types is done.
- If an ANSI-prototype declaration is present with an old-style definition, then the declaration must use promoted variable types (**int** or **double** instead of **char**, **short**, or **float**).

Auto Aggregate Initialization

An AGGREGATE type is a generic term for a complex C type, and the term generally refers to structures and arrays. ANSI allows a feature known as AUTO-AGGREGATE INITIALIZATION, which means essentially that the aggregate types (structures and arrays) may be initialized as local (auto) variables. Two examples of auto-aggregate initialization would be:

```
struct st_tag {
    int a, b;
};

void funkier_func (void)
{
    struct st_tag a_struct = {0, 1};
    char name[] = "mike";
    .
    .
    .
}
```

The variables **a_struct** and **name** are the auto-aggregates that are initialized. Auto-aggregate initialization is not supported by K&R and is not implemented on many UNIX compilers.

Wide Strings and Literals

Wide literals and strings are used to address the problem of character sets which are too large to fit within the constraints of an 8-bit **char**. Wide character constants and string literals are specified by preceding the argument with an **L** or **L**, and are considered to be of type **wchar_t** (defined in **stddef.h**). The **wchar_t** type is defined by Aztec C68K to simply be of type **char**. For example:

```
wchar_t *cp = L"ABC";
```

is a declaration of wide string literal pointer. As is implied above, wide strings and literals are currently implemented exactly the same as normal characters and are included only to provide full ANSI compatibility.

Structure Assignment and Structure Passing

Assignment of one structure to another is supported. In the example:

```
struct st {
    int st_type;
    char *st_name;
} st_first = {5, "Hey!"};

struct st st_second;
void func (void)
{
    st_second = st_first;
}
```

the structure `st_second` will take on the value 5 for `st_second.st_type`, and Hey! for `st_second.st_name`.

Passing of full structures to a function is also legal; the compiler simply copies the contents of the structure onto the stack. A function can also return a structure as its value.

Trigraphs

TRIGRAPHs are three-character alternate representations of certain non-alphanumeric characters. They exist primarily for systems which cannot input some of the less common non-alphanumeric characters directly from the keyboard. The supported trigraphs are:

<i>Trigraph</i>	<i>Character it represents</i>
-----------------	--------------------------------

??=	#
??([
??/	\
??)]
??'	^
??<	{
??!	
??>	}
??-	~

By default, Aztec C68K does not parse trigraphs. However, tri-graph parsing may be turned on by means of a compiler switch.

Escape Sequences

The following character escape sequences are supported by Aztec C68K:

\'	single-quote.
\"	double-quote.
\?	question-mark.
\\"	backslash.
\a	Alert - flashes the Amiga screen briefly.
\b	Backspace - Moves the cursor back one space.
\f	Form feed - Move to the next physical page.
\n	New line - Move to the start of the next line.
\r	Carriage return - Move to the start of the current line.
\t	Horizontal tab - Move to the next tab position.
\v	Vertical tab - Move to the same horizontal position on the next line.
\x	Hex integer - the digits following the \x define a hexadecimal constant.
\digits	Octal integer - the digits following the \ define an octal constant.

Long Character Constants

Aztec C68K recognizes long character constants. The following code will put the ASCII values for **a**, **b**, **c**, and **d** into the long value **l**:

```
long l;  
  
l = 'abcd';
```

Note that this is different than a string, in that each letter is considered to be a character-sized constant, and there is no null appended to the end.

Register Variables

The Aztec C compiler supports a total of nine register variables. Specifically, six data registers (**d2-d7**, for general non-pointer data) and three address registers (**a2, a3, a6**, for pointers) are reserved for register variables.

The compiler can automatically allocate register variables based on the relative usage of each variable, without an explicit **register** declaration. This behavior is entirely optional and is turned off by default.

The following types may be declared as register variables:

- **char, short, int, long, pointer, float, double**

In addition to these, the unsigned counterparts of the above may also be declared as **register**.

Sizeof and 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. **size_t** is always an integral type, and should be large enough to contain the size of the largest data element addressable by the compiler. On the Aztec C68K system, **size_t** is defined to be a **long**. Because **size_t** returns a **long**, data objects in the large data model may be of any size (in the small data model, total data is limited to 64K).

Symbol Names

All symbol names, both of internal and external linkage, have 31 significant characters. Symbols with global scope have a "_" prepended to their names internally. This "_" is invisible at the C source level, but is necessary when accessing C variables from assembly language.

Preprocessor Features

Predefined Symbols

The pre-defined preprocessor symbols **_FILE_**, **_LINE_**, and **_FUNC_** are fully supported, as per ANSI.

- `_FILE_` is a string which expands to the name of the current file being compiled
- `_LINE_` expands to an integer representing the current line number
- `_FUNC_` expands to a string which specifies the current function being compiled.

For example, consider the following C file called `test.c`:

```
void
funky_func (void)
{
    printf ("File = '%s'", _FILE_);
    printf ("Line = %d", _LINE_);
    printf ("Function = '%s'", _FUNC_);
}
```

The function `funky_func` will produce this output when it is called:

```
File = 'test.c'
Line = 5
Function = 'funky_func'
```

There are several symbols specific to Aztec C68K that are always predefined by the compiler. These are: `MPU68000`, `MCH_AMIGA`, and `AZTEC_C`. Also, a symbol named `VERSION` is defined and set to 500 (version 5.00).

Other symbols (also specific to Aztec C) that may be defined are:

SYMBOL *Defined when...*

<code>_INT32</code>	Integers are 32-bits in length
<code>_LARGE_CODE</code>	The large code model is being used.
<code>_LARGE_DATA</code>	The large data model is being used.

String Concatenation

String-concatenation is a pre-processor feature which allows two or more strings which occur next to each other in C source to be concatenated together and treated as one string. For example:

```
#define ERROR "ERROR: "
    printf (ERROR "Hey you! "
    "A system error has occurred!!");
```

In this example, the `printf` call would print out:

```
ERROR: Hey you! A system error has occurred!!
```

There are two useful applications for string concatenation:

- Long strings may be separated into several short strings. This is often desirable to keep the indentation consistent within the body of a function. This is why `Hey you!` and `A system error has occurred!!` are split—to preserve the indentation of the surrounding program, thereby making it easier to read.
- `#defines` may be used for common prefixes for strings. In the above example, the string `ERROR:` is assigned to a macro named `ERROR`, which might be a standard prefix for error messages.

Line Continuation

Logical lines that cannot fit on one line may be continued by use of the line continuation character, `\` (a backslash), being placed at the end of the physical line. For example, the following statements define a character array with the contents `abcdef`:

```
char an_array[] = "ab\
cd\
ef";
```

It is useful to note that, in most cases, string concatenation is a preferable method to line continuation. This is because string concatenation can preserve a program's indentation, whereas line continuation cannot. The tabs or spaces used for indenting would show up in the string.

ASSEMBLER

4

Chapter 4 - Assembler

The Manx Aztec C Assembler, **as**, translates assembly language source statements into relocatable object code.

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 Manx Aztec C Linker ,ln, before it can be executed. At linkage time it may be combined with other object files and run time 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. The cc command 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).

Operating Instructions

The assembler is started by entering the command line

as [-options] filename

where **[-options]** specify optional parameters and *filename* is the name of the file to be assembled.

The assembler is also invoked by the C Compiler to assemble its output file.

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.

EXECUTION ENVIRONMENT

Processor Support

The assembler supports the MC68010, MC68020, MC68030, and the MC68881, MC68851, instruction sets and addressing modes, in addition to those of the MC68000.

The assembler defaults to assuming that only the MC68000 instructions are valid. The **machine**, **mc68881**, and **mc68851** directives enable and/or disable the additional instructions and addressing modes.

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:

as 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 **.o**.

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.

```
as -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. It also searches a user-specified sequence of directories for such files, thus allowing program source files and header files to be contained in different directories.

Option **-i** and the **INCLUDE** environment variable define the directories the assembler searches for **include** files.

The assembler automatically searches only the current directory for an include file if the following conditions are met:

- (1) the assembler is started without option **-i**
- (2) **INCLUDE** is not an environment variable, and
- (3) the **include** statement does not specify the drive and/or directory containing the file.

If an **include** statement specifies either the drive or directory, only that location is searched 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

```
as -isys:db/include prog1
```

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

INCLUDE Environment Variable

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

The INCLUDE variable consists of the names of the directories to be searched, separated by semicolons. For example, the following command creates the INCLUDE environment variable, defining three directories to be searched:

```
set INCLUDE=work:include;work::sys:include
```

These directories are (1) the **include** directory on the **work:** volume (2) the root directory on the **work:** volume, and (3) the **include** directory on the **sys:** volume.

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 INCLUDE environment variable in the order listed

MODULE MEMORY MODEL

Options **-c** and **-d** instruct the assembler to use the large code and large data models, respectively. For a discussion of these models, see the Compiler chapter of this manual.

Assembler Options

SUMMARY OF OPTIONS

- a Forces total size of code and data to be aligned to a 4-byte boundary instead of the default 2-byte boundary.
- c Makes large code the default code memory model. May be overridden by the **near code** and/or **far code** directives.
- d Makes large data the default data memory model. May be overridden by the **near data** and/or **far data** directives
- ename[=val] Creates 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.
- iarea Defines an area to be searched for files specified in an **include** statement.
- l Generates an assembly listing file.
- m Metacomco compatibility option.
- n Does not optimize object code.
- o *filename* Specifies name of output object module.
- v Verbose option. Generates memory usage statistics.
- ZAP** Used primarily by the Aztec C compiler to direct the assembler to delete the input file after processing.

DESCRIPTION OF OPTIONS

Option -c

If you use this option, the assembled module uses the large code memory model.

Option -d

If you use this option, the assembled module uses the large data memory model.

Option -o *filename*

If you use this option, the assembler sends the object code to *filename*. If option **-o** is not specified, the assembler sends the object code to a file whose name is derived from that of the assembler source file (by changing the extension to **.o**). In this case, the file is placed in the directory containing the source file.

Option -i

The **-i** option specifies where files referred to by the **include** directive are kept.

If you use option **-i**, the assembler searches a specified area for files included in the source code.

The name of the area should immediately follow the **-i** with no intervening spaces. For example, the following defines the directory **source/inc** on volume **sys:** as an area to search for include files:

-isys:source/inc

For more details, see page 4-4.

Option -l

If you specify option **-l**, the assembler generates a listing file. The name of the file to which the listing is sent is derived from that of the source file by changing the extension to **.lst**. The listing file is placed in the directory containing the source file.

Source Program Structure

There are four types of Assembler statements:

- (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 used for a comment may alternatively begin with an asterisk.

EXECUTABLE INSTRUCTIONS

Executable instructions have the general format:

label operation operand

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. Note: C prefixes all external symbols with an underscore. For example, a C data object, int i, will appear as _i at the assembly level and should be referenced as such.

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 comprises register mnemonics, symbols, constants, or arithmetic combinations of symbols or constants. In addition certain 68020 instructions require special operand syntax.

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.

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:

@123 or @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'`

Register mnemonics include the data registers `d0` through `d7`; the address registers `a0` through `a7`; `sp` (same as `a7`) the stack pointer; `pc` the program counter (forces PC relative mode); `sr` the status register; `ccr` the condition code register ; and the user stack pointer `usp`.

The assembler supports addition (+), subtraction (-), multiplication (*), division (/), shift right (>), shift left (<), unary minus, and (&), or (!). It also supports inclusive or (!), exclusive or (^), bitwise not (~), and modulo (//).

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

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.

The default setting of `off` treats a blank as the end of the operand field.

The `blanks on` setting 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.

cseg

The assembly code following this directive is output into the code segment of the program output file.

dseg

The assembly code following this directive is placed in the initialized data segment of the program file.

dc - Define Constant

[label]	dc.b	<i>value [value, value ...]</i>
[label]	dc	<i>value [value, value ...]</i>
[label]	dc.w	<i>value [value, value ...]</i>
[label]	dc.l	<i>value [value, value ...]</i>
[label]	dc.b	" <i>string</i> "

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.

For command programs, a value can contain a reference to a memory location whose address will not be known until the program is loaded into memory. In this case, an item for this value will be added to the program's relocation table. When the program is loaded, the field containing this value will be set to the correct value.

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 2 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	<i>size [value]</i>
[label]	dcb	<i>size [value]</i>
[label]	dcb.w	<i>size [value]</i>
[label]	dcb.l	<i>size [value]</i>

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 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

<i>[label]</i>	ds.b	<i>size</i>
<i>[label]</i>	ds	<i>size</i>
<i>[label]</i>	ds.w	<i>size</i>
<i>[label]</i>	ds.l	<i>size</i>

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).

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. The entry point must be in the first 32K of the root segment.

end

This directive defines the end of the source statements.

equ

label **equ** *expression*

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

equir

[label] **equir** *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

This directive causes the assembler to generate code for the large code memory model. Absolute addressing will be used with fixups being done by the startup code.

far data

This directive causes the assembler to generate code for the large data memory model. Absolute addressing will be used with fixups being done by the startup code.

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, the **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 the 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.

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

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 *symbol*[,*symbol*..]

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 does not.

reg

label reg *register list*

This directive assigns the value of the register list to the label. Forward references are not allowed. A 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

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 direc-

tive. 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.

Interfacing With C

Interfacing 68000 assembly language routines with C is relatively easy. The linkage conventions are straightforward and simple.

REGISTER CONVENTIONS

It is the responsibility of an assembly language subroutine to preserve the values in registers **a2** through **a5** and **a7**, as well as registers **d4** through **d7**.

Register **d0** through **d3** and **a0**, **a1** and **a6** are available as work registers. There is no need to preserve the values of work registers for other routines.

Register **d0** contains the return value of the subroutine if the return type is non floating point. If the return value is floating point, then the return value is in the register pair **d0-d1**.

ARGUMENTS TO SUBROUTINES

Upon entry register **a7** (**sp**) points to the stack. The first item on the stack is a 32 bit absolute return address. The second item on the stack is the first argument to the subroutine, followed by the second, and so on. Arguments to C subroutines are passed by value. Therefore character, integer, long, and floating point arguments are copied onto the stack by value. If the functions are not prototyped, character items are promoted to type **int** before being pushed on the stack, otherwise character items are passed normally.

Variable Names

The compiler attaches a leading underscore to all variable names. Therefore, in order to access these labels or names from within an assembly file, the programmer must prefix the labels with an underscore. The labels must also be declared as **public** or followed by a **#** which makes them public.

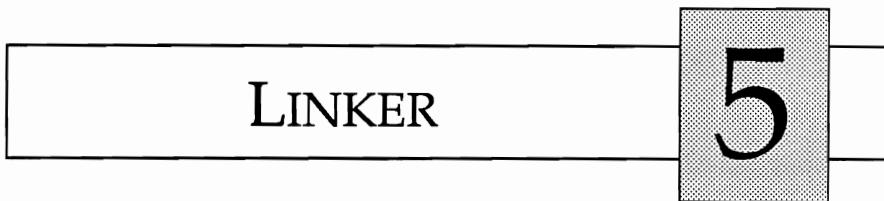
Returning from a C Function

To return from a C function, it is necessary to do the following:

- restore the values of registers d4 through d7 as well as registers a2 through a5 and a7 to the values they had at entry to the routine;
- to place the return value, if any, in register d0 or in d0 + d1 if floating point; and
- to execute an `rts` instruction.

It is not necessary to restore values to registers that were not changed.

Upon return to the calling routine, the stack still contains the arguments that were passed to the subroutine. The first argument is on the top of the stack.



Chapter 5 - Linker

The Manx linker creates executable programs by linking together the pieces of the program that are compiled and assembled and are in relocatable object format.

The linker has two functions:

- It ties together the pieces of a program that were compiled and assembled separately
- It converts the linked pieces to a format that can be loaded and executed.

The Manx Assembler must first create the pieces.

From those pieces, the linker can create several types of executable objects:

- **CLI Programs**
- **Workbench Programs**
- **Drivers**, which other programs call to access devices

Introduction to Linking

For those with limited exposure to linkers, this section introduces linking in general and the Manx linker in particular. If you are experienced or wish to skip this section, go right to “Using the Linker”.

LINKING Hello.o

It is very unusual for a C program to consist of a single, self-contained module. Consider a simple program that prints

```
hello, world
```

using the function, `printf()`. The terminology here is precise: `printf()` is a function and not an intrinsic feature of the language. It is a function that you might have written but did not have to, because Aztec C already provides it in the file, `c.lib`. This file is a library of all the standard I/O functions. It also contains many support routines that are called in the code generated by the compiler. These routines aid in integer arithmetic, operating system support, etc.

When the linker sees that a call to `printf()` is made, it pulls the function from the library and combines it with the `hello, world` program. The link command looks like this:

```
ln hello.o c.lib
```

or

```
ln hello.o -lc
```

When `hello.c` is compiled, calls are made to some invisible support functions in the library. So linking without the standard library causes some unfamiliar symbols to be undefined.

Almost all programs need to be linked with some variant of `c.lib`.

THE LINKING PROCESS

Since the standard library contains only a limited number of general purpose functions, all but the most trivial programs are certain to call

user-defined functions. It is up to the linker to connect a function call with the definition of the function somewhere in the code.

In the example given below, the linker finds two function calls in **file1**. The reference to **func1** is "resolved" when the definition of **func1** is found in the same file. However, the reference to **func2** is unresolved because **func2** is found in another file, e.g., **file2.o**. Therefore, the following command

```
ln file1.o c.lib
```

causes an error indicating that **func2** is an undefined symbol. To be successful, the linkage must include **file2.o** as follows:

```
ln file1.o file2.o c.lib
```

Example

File 1

```
main()
{
    func1();
    func2();
}
{
    func1()
{
    return;
}
```

File 2

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

Libraries

A library is a collection of object files put together by a librarian. Libraries intended for use with the linker must be built with the Manx librarian, **lb**. This utility is described in the **Utilities** chapter.

The linker also recognizes AmigaDos libraries (**Amiga.lib**) See the discussion of these in the appropriate chapter of your *Aztec C Library Manual*.

All the object files specified to the linker are pulled into the linkage; they are automatically included in the final executable file. However, when a library is encountered, it is searched. Only those modules in the library that satisfy a previous function call are pulled in.

The **CLIB** environment variable, used to specify where libraries may be found, supports multiple entries when they are separated by an exclamation point (!). For example, if libraries are in both the **sys2:lib** and **ram:** directories, then **CLIB** would be defined like this:

```
set CLIB=sys2:lib!ram!
```

The linker automatically adds an **.o** extension to files that have no extension. It checks all directories defined in the **CLIB** environment variable and then the current directory .

EXAMPLE

Consider the "hello, world" example. Having looked at the module, **hello.o**, the linker has built a list of undefined symbols.

This list includes all the global symbols referenced but not defined. Global variables and all function names are considered to be global symbols.

The list of undefineds for **hello.o** includes the symbol **printf()**. When the linker reaches the standard library, this is one of the symbols it will be looking for. It discovers that **printf()** is defined in a library module whose name also happens to be **printf()**. (There is no relation between the name of a library module and the functions defined within it.)

The linker pulls in the **printf()** module in order to resolve the reference to the **printf()** function.

Files are examined in the order in which they are specified on the command line. So the following linkages are equivalent:

```
ln hello.o  
ln c.lib hello.o
```

Since no symbols are undefined when the linker searches **c.lib** in the second line, no modules are pulled in. It is good practice to leave all libraries at the end of the command line, with the standard library last of all.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant. The linker searches a library 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 undefineds. The linker does not search the library twice to resolve any references that remain unresolved. A common error lies in the following situation:

Module of Program	References (function calls)
<code>main.o</code>	<code>getinput, do_calc</code>
<code>input.o</code>	<code>gets</code>
<code>calc.o</code>	<code>put_value</code>
<code>output.o</code>	<code>printf</code>

Suppose we build a library to hold the last three modules of this program. Then our link step looks like this:

```
ln main.o proglb.lib c.lib
```

But it is important that `proglb.lib` is built in the right order. Assume that `main()` calls two functions, `getinput()` and `do_calc()`. `getinput()` is defined in the module `input.o`. It in turn calls the standard library function, `gets()`. `do_calc()` is in `calc.o` and calls `put_value()`. `put_value()` is in `output.o` and calls `printf()`.

The following paragraphs describe what happens at link time if `proglb.lib` is built in the order shown below:

```
proglb.lib:  input.o
              output.o
              calc.o
```

After `main.o`, the linker has `getinput` and `do_calc` undefined (as well as some other obscure functions in `c.lib`). Then it begins the search of `proglb.lib`. It looks at the library module, `input`, first. Since that

module defines `getinp` module defines `getinput`, that symbol is taken off the list of undefineds. But `gets` is added to it.

The symbols **do_calc** and **gets** are undefined when the linker examines the module **output**. Since neither of these symbols is defined there, that module is ignored. In the next module, **calc**, the reference to **do_calc** is resolved but **put_value** is a new undefined symbol.

The linker still has **gets** and **put_value** undefined. It then moves on to **c.lib**, where **gets** is resolved. But the call to **put_value** is never satisfied. The error from the linker looks like this:

```
Undefined symbol: _put_value
```

This means that the module defining **put_value** is not pulled into the linkage. The reason, as we saw, was that **put_value** was not an undefined symbol when the **output** module was searched. This problem would not occur with the library built this way:

```
proglib.lib:    input.o
                calc.
                output.o
```

The **ord** command (described in the Utilities chapter) calculates a workable order for a collection of **.o** files.

Occasionally it becomes difficult or impossible to build a library so that all references are resolved. In the example, the problem could be solved with the following command:

```
ln main.o proglib.lib proglib.lib c.lib
```

The reason this is not the most satisfactory solution is that the linker must search the library twice, thus lengthening linking time.

The most common cause of cycles is illustrated by the following example:

```
f.c:          g.c:
int a;
f()
{
    g();
}
extern int a;
g()
{
    a = 4;
}
```

Here, **f()** invokes **g()** so file **g.o** should follow **f.o** in the library. However, **g()** references **a**, so **f.o** must follow **g.o**.

The solution to this problem is quite straightforward, however:

f.c	g.c:	a.c:
extern int a;	extern int a;	int a;
f()	g()	
{...g();...}	{...a = 4; }	

By moving the definition of **a** to a separate file, the order **f.o**, **g.o**, **a.o** allows all references to be resolved.

When there is no way to remove a cycle, the **ord** utility announces that fact and chooses **a.o** to break the cycle. The output of **ord** then contains duplicates.

You can then try using the splitting technique above, or use the output from **ord** with **lb** to construct a library with duplicates of some modules.

Using the Linker

As mentioned in the introduction to this chapter, the linker can create several types of executable programs. Much of the actual use of the linker is the same, regardless of the type of program generated.

STARTING THE LINKER

Start the linker by entering:

ln [-options] file1 file2 ...

where **[-options]** are linker options and **file1 file2 ...** are the names of files containing object modules and libraries of object modules.

For example, the most basic linker command which creates an executable "hello, world" program by linking object code in **hello.o** with modules in **c.lib** is

```
ln hello.o c.lib
```

This creates a command program in the file **hello**.

INPUT FILES

The linker scans the input files in the order in which they are specified on the command line. If a file contains a single object module, the module is automatically included in the program being built.

If an input file is a library of object modules, the linker makes one pass through the library, looking for modules containing a function or global variable that has been called by an already-included module and that is not in any already-included module. When such a module is found, it is included in the program being built.

In other words, when scanning a library of object modules, the linker only includes needed modules in the program it is building.

A filename passed to the linker has the standard AmigaDos path format, i.e., it consists of an optional volume name, optional path to the directory containing the file, and the filename itself. The volume defaults to the current volume and the directory to the current directory.

EXECUTABLE FILES

The name of the file to which the linker writes the executable program can be specified using linker option **-o**. If this option is not used, the linker derives the name of the executable file from that of the first object module listed on the command line, by deleting the extension. In the default case, the executable file is placed in the same directory in which the first object module is located.

For example, the following instruction links **main.o**, **menu.o**, and **add.o** with the Manx library **c.lib**, all of which are in the current

directory, and sends the executable program to the file named **main** in the current directory:

```
ln main.o menu.o add.o c.lib
```

And the following links the same modules, placing the output in **sys:bin/myprog**

```
ln -o sys:bin/myprog main.o menu.o add.o c.lib
```

LIBRARIES

Aztec C provides the following basic libraries:

c.lib
m.lib

All programs must be linked with **c.lib**. In addition to all the nonfloating point functions described in the library chapter, **c.lib** contains internal functions that are called by compiler-generated code, such as functions to process switch statements.

m.lib contains the transcendental floating point functions described in the library chapter, such as **sin()**, and versions of the functions **printf()**, **fprintf()**, **sprintf()**, **scanf()**, **fscanf()**, and **sscanf()**. A program that calls any of these functions must be linked with **m.lib** in addition to **c.lib**.

Otherwise, a program need not be linked with **m.lib**. In particular, if it performs floating point operations without calling any of these functions, it does not need to be linked with **m.lib**.

When a program calls a **printf()**- or **scanf()**-type function to perform a floating point conversion, the linker must search **m.lib** before it searches **c.lib**. The reason for this is that there are two versions of these functions: the ones that support floating point are in **m.lib**; the others that do not are in **c.lib**. If **c.lib** is searched before **m.lib**, then when a program that requires the floating point versions is linked, the nonfloating point version of a function will be used and the program will misbehave.

For additional information on the other libraries included with your Aztec C development package refer to the **Library Overview** chapter of the *Aztec C Library Manual*.

SUMMARY OF LINKER OPTIONS

The linker options are summarized as follows:

- +a Forces each module to be aligned on a long word boundary. For most applications this is not necessary and will only make the program larger. However, in certain cases (i.e. BCPL pointers) it is necessary for long word alignment.
- +c[cdb] Forces loading into chip memory (i.e. standard built-in memory) of the program sections whose identifying letters follow the +c.
- +f Forces loading into fast memory (i.e. external add-on memory) of the program sections whose identifying letters follows the +f.
- f *file* Reads command arguments from *file*.
- g Tells the linker to generate a source level debugger output file. This file has the same root name as the program output file and the extension .dbg. This file is used by the optional Aztec C Source Level Debugger.
- +l This option tells the linker that the following Amiga object modules are to be treated as though they are libraries until the next +l is encountered. This option acts as a toggle and may be used several times in a single link line.
- l *name* Library, e.g., -lc for c.lib
- m Disables linker warnings about object module symbols overriding library symbols.

- o *file* Specifies the name *file* to which the executable program will be sent. If not given, the name of the file is the same as that of the first input file, with the extension deleted.
- +o[i] Places the following object modules in code segment *i*. If no number is specified, use the first empty segment. If the segment already exists, the modules are added to its end.
- q Turns off source level debugging file generation. (See -g.)
- +q Disable the module by module display while linking. The linker displays the names of the various modules as it reads them. This option causes the linker to suppress the display.
- +sss +ss+s *val* Specifies four different models for scatter loading, ranging from one big hunk to every module in its own hunk.
- t Generates an ASCII symbol table file.
- w Generates a Wack-readable symbol table . This table is also usable by db (see Chapter 7, DB Debugger.)
- v Specifies verbose link.

DETAILED DESCRIPTION OF OPTIONS

Option -l

Option -l provides a convenient way to link programs located in one directory with libraries located in another.

Option -l is immediately followed by the partial name of a library file. The linker builds the complete name by prefixing it with the string associated with the environment variable CLIB and appending the string .lib to it.

For example, if the libraries are located in the directory **sys:lib**, then set CLIB to:

```
sys:lib/
```

and the object module **prog.o** could be linked with the libraries **mylib.lib**, **m.lib**, and **c.lib** using the command

```
ln prog.o -lmylib -lm -lc
```

Option **-f**

This option causes the linker to continue reading options filename from a file; when done, it then continues reading arguments from the command line. The name of the file follows option **-f**.

For example, the following command links **prog.o** with **sub1.o**, **sub2.o**, ..., **m.lib** and **c.lib**; it reads some filenames from the file **prog.lnk**:

```
ln -f prog.lnk -lm -lc
```

where **prog.lnk** contains

```
-o prog.out
sub1.o sub2.o
sub3.o
sub4.o
```

The contents of files read by the **-f** option should contain only the names of object modules and libraries to be linked. It should not contain any additional linker options.

Option **+**

This flag specifies that the following Amiga object modules are really Amiga libraries until the next option **+l** is detected. The Linker automatically detects that a module is in Amiga format. However, because an Amiga object library is simply a concatenation of a number of modules, it is necessary to tell the Linker that the following module is a library.

Otherwise, the Linker adds all the modules in the file to the program. For example, if modules **am1.o** and **am2.o** are Amiga libraries, they can be linked in as follows along with the program **prog.o**:

```
ln prog.o +l am1.o am2.o +l -lc
```

Option **-m**

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, the warnings are suppressed.

Option **-o**

Option **-o** 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**:

```
ln -o progout prog.o c.lib
```

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

Option **+o[i]**

The linker uses this option to handle segmentation (overlay). The executable code in the object modules that follow the option is placed in code segment *i*. If *i* is not specified, use the first empty segment number. If the segment already exists, append the code to its end. When segments are used, the linker generates a reference to the symbol **.segload** which is defined in a library module, **segload.o**. This module must be in the root segment for the program to function properly. The module is also available directly in the **lib** directory.

Segments are loaded into memory as needed and remain in memory until explicitly removed by the program. The program does this by calling the **freeseg()** routine with the address of a function that is in the segment to be unloaded.

Option +s, +ss, +sss

The Linker supports four models of scatter loading. Default model is no option and specifies code in one hunk. Linker option **+s** places each file in a separate hunk. This means that modules concatenated together into a single file and all modules pulled from a single library are placed together in the same hunk. Linker option **+ss** adds files to a hunk until the size of the hunk reaches 8K. The linker then starts a new hunk. Finally, Linker option **+sss** puts each module into its own individual hunk.

The Linker generates a **jmp** instruction at the beginning of Hunk 0 if there has been an entry point defined and the entry point is not already at the beginning of Hunk 0.

Option -t

The **-t** option creates an ASCII file that contains a symbol table for the linker. This file is just a text file which lists each symbol with a hexadecimal address. This address is either the entry point for a function or the location in memory of a data item. A perusal of this file will indicate which functions were actually included in the program.

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 in the **Program Organization** section of the **Technical Information** chapter.

Option -w

The **-w** option creates a disk file that contains a symbol table for the linker, in the format expected by Wack.

Option -v

The **-v** option causes the linker to send a progress report of the linkage to the screen as each input file is processed. This is useful in tracking down undefined symbols and other errors which may occur while linking.

Options +c and +f

There are two types of memory on an Amiga: 'chip' memory, which is inside the Amiga; and 'fast' memory, which is external to the Amiga. By default, the three sections of a program (code, initialized data, and uninitialized data) are loaded into whatever memory is available. The +c and +f options allow you to force selected sections of a program to be loaded into chip and fast memory, respectively. Program sections are identified to an option by appending a letter to the option. These letters, and the sections to which they refer are:

- c Executable code.
- d Initialized data.
- b Uninitialized data (bss).

If an option is used without appending any letters to it, then all the program's sections are forced to be loaded into the specified type of memory.

For example:

```
ln +cdb +fc ....
```

will force a program's initialized and uninitialized data to be loaded into chip memory and the program's executable code to be loaded into fast memory. Note that if there is no extra memory, the program will not load.

More normally, you would use:

```
ln +cdb ....
```

which will place the program's initialized and uninitialized data into chip memory and its executable code into whatever memory is available.

Finally:

```
ln +c ...
```

will force the program's initialized data, uninitialized data, and executable code to be loaded into chip memory; it is equivalent to +ccdb.

Option -g

The **-g** option is for use with the Manx Source Level Debugger (**sdb**). This option, combined with the compiler's **-bs** option, will generate a file with the **.dbg** suffix in the current directory. This **.dbg** file provides information necessary for use with **sdb**.

For example:

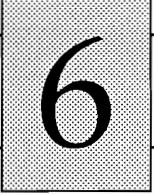
To compile and link the "hello, world" program and generate the **.dbg** file, you would use the following commands:

```
cc -bs hello.c  
ln -g hello.o -lc
```

This will create the file **hello.dbg** in the current directory.

Using the **-g** option without having SDB installed will have no effect on the final program execution.

UTILITIES



Chapter 6 - Utilities

This chapter describes the Aztec C utility programs, in alphabetical order.

NAME	arcv - text dearchiver
SYNOPSIS	arcv <i>arcvfile</i> [<i>dest_dir</i>]
DESCRIPTION	arcv is the counterpart to the utility mkarcv and is designed to extract files that were originally archived with mkarcv . arcv will take the archive <i>arcvfile</i> and extract all of the text files stored within it. The original archived file is left intact. The extracted files are placed in the directory specified by <i>dest_dir</i> ; if <i>dest_dir</i> is not specified, the files are placed in the current directory. If subdirectories have been included in the archive, arcv will create these and place the dearchived text files in them.
EXAMPLE	For example, if 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.

NAME	adump - Amiga object module dump utility
SYNOPSIS	adump [-cds] file
DESCRIPTION	adump displays information about <i>file</i> , which contains an object module or load module that has been generated by the Amiga Assembler or Linker.
	adump always lists the length of each of the module's blocks and the offset of each block. The options to adump cause it to display the following additional information:
Option	Meaning
-c	Display code block contents in hex
-d	Display data block contents in hex
-s	Display symbol block contents (i.e., symbol names). Symbol blocks are found in executable files if debugging information has been included.

NAME	cmp - file comparison utility
SYNOPSIS	cmp [-l] <i>file1</i> <i>file2</i>
DESCRIPTION	cmp compares two files on a character-by-character basis. When it finds a difference, it displays a message, giving the offset from the beginning of the file. If option -l is not specified, the program will stop after the first difference, displaying a message in the format:
	Files differ: character 10
	If option -l is specified, cmp lists all differences, in the format: <i>decimal-offset hex-offset: f1-value f2-value</i> where <i>f1</i> and <i>f2</i> are files.
EXAMPLES	cmp <i>otst</i> <i>ntst</i> Files differ: character 10 and cmp -l <i>otst</i> <i>ntst</i> 10 a: 00 45 100 64: 1a 23

NAME cnm - display object file info

SYNOPSIS cnm [-sol] *file* [*file* ...]

DESCRIPTION cnm 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 lb librarian. However, cnm does not support Amiga object module format.

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

```
cnm sub1.o 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.o** to the display and to the file **dispfile**:

```
cnm sub1.o  
cnm > dispfile sub1.o
```

The first line listed by cnm 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.

OPTIONS

Option List

- s Display size only
- l Display each symbol on a separate line
- o Prefix symbols with filename

If **cnm** 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.

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

When **cnm** displays information about the module named symbols, option **-l** tells **cnm** to display each symbol's information on a separate line and to display all of the characters in a symbol's name; if this option is not used, **cnm** displays the information about several symbols on a line and only displays the first eight characters of a symbol name.

Option **-o** tells **cnm** 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 **cnm** in combination with **grep**. For example, the following commands display all information about the module **perror** in the library **c.lib**:

```
cnm -o c.lib tmp  
grep perror tmp
```

Symbol Format

cnm 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, **cnm** 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.

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, that is, callable from other programs, and hence have type `Pg`.

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`.

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.

bs

The symbol is in the uninitialized 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.

G1

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

The compiler generates **G1** symbols for nonstatic, uninitialized variables that are declared outside all functions and that are not dimensionless arrays.

The assembler generates **G1** symbols for variables declared using the global directive that have a nonzero size.

NAME	du - disk usage
SYNOPSIS	du <i>directory1</i> [<i>directory2...</i>]
DESCRIPTION	<p>The du command displays the disk usage information for the specified directory. The command automatically operates upon any subdirectories. The result is specified for each directory in blocks. The blocks from a subdirectory are added to the parent directory's total.</p> <p>If no argument is specified, it defaults to the current directory.</p>

NAME	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. <i>file1, file2, ...</i> are the names of the files to be displayed. + <i>n</i> specifies the offset into the files where the display is to start and defaults to the beginning of the file. If + <i>n</i> is followed by a period, <i>n</i> 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

NAME	lb - object file librarian																								
SYNOPSIS	lb <i>library [options] [mod1 mod2 ...]</i>																								
DESCRIPTION	lb is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.																								
ARGUMENTS AND OPTIONS	Library Argument lb acts upon a single library file. The first argument to lb (<i>library</i> , in the synopsis) is the name of this file. The filename extension for <i>library</i> is optional; if not specified, it is assumed to be .lib. Options Argument There are function code options and qualifier options. These options are summarized and then described in detail. FUNCTION CODE OPTIONS lb performs <i>one</i> function at a time on the specified library. The functions that lb can perform, and the corresponding option codes, are:																								
	<table><thead><tr><th>Function</th><th>Code</th></tr></thead><tbody><tr><td>create a library</td><td>(no code)</td></tr><tr><td>add modules to a library</td><td>-a,-i, -b</td></tr><tr><td>list library modules</td><td>-t</td></tr><tr><td>move modules in a library</td><td>-m</td></tr><tr><td>replace modules</td><td>-r</td></tr><tr><td>delete modules</td><td>-d</td></tr><tr><td>extract modules</td><td>-x</td></tr><tr><td>ensure module uniqueness</td><td>-u</td></tr><tr><td>define module extension</td><td>-e</td></tr><tr><td>read function from file</td><td>-f</td></tr><tr><td>help</td><td>-h</td></tr></tbody></table>	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
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, **lb** 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 **lb** 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 **lb** 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 **lb** is to use. For some functions, **lb** 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, **lb** will assume that it is **o**. You can explicitly define the default module extension using Option **-e** (see pages 6-22.)

Reading Arguments from Another File

lb 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 of **lb**

This section describes the basic features of **lb**. The basic points you need to know about **lb** 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 **lb** arguments from a file.

HOW TO CREATE A LIBRARY

A library is created by starting **lb** 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 **lb** 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**:

```
lb exempl.lib obj1.o obj2.o
```

Making use of the **lb** assumptions about filenames for which no extension is specified, the following command is equivalent to the above command:

```
lb exempl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by **lb**. In this case, all the modules in the input library are copied into the new library.

When **lb** creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, **lb** 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, **lb** 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 THE NAMES OF MODULES IN A LIBRARY

To list the names of the modules in a library, use the **lb** option **-t**. For example, the following command lists the modules that are in **exmpl.lib**:

```
lb exempl.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 **lb** 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

lb newlib mod1 oldlib.lib mod2 mod3

the contents of the newly-created **newlib.lib** will be:

mod1 sub1 sub2 sub3 mod2 mod3

The **ord** utility program can be used to create a library whose modules are optimally sorted. For information, see the **ord** description later in this chapter.

GETTING LB ARGUMENTS FROM A FILE

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to **lb** on a single command line. In this case, the **lb -f filename** feature can be of use: when **lb** 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

lb -f build

causes **lb** to get its arguments from the file **build**, which causes **lb** to create the library **exmpl.lib** containing **obj1** and **obj2**.

Arguments in a **-f** file can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a **-f** file can be on separate lines, if desired.

The **lb** command line can contain multiple **-f** arguments, allowing **lb** 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**:

```
lb exmpl -f arith.inc -f input.inc  
-f output.inc
```

A -f file can contain any valid **lb** argument, except for another -f. That is, -f files cannot be nested.

ADVANCED **lb** FEATURES

In this section we describe the rest of the functions that **lb** can perform. These primarily involve manipulating selected modules within a library.

ADDING MODULES TO A LIBRARY

lb 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** function 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 **lb** command that selects the **add** function, 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

```
lb exmpl -i obj2 mod1 mod2
```

adds the modules in the files **mod1.o** and **mod2.o** to **exmpl.lib**, placing them before the module **obj2**. The resultant **exmpl.lib** looks like this:

```
obj1  mod1  mod2  obj2  obj3
```

ADDING MODULES AT THE BEGINNING OR END OF A LIBRARY

Options **-b+** and **-a+** tell **lb** 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**:

```
lb 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:

```
lb 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 <i>target</i>
-ma target	move modules after the module <i>target</i>
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the **lb** 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 **lb** command.

DELETING MODULES

Modules can be deleted from a library using option **-d**. The command for deletion has the form

lb 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:

```
lb 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 `lb` command to replace a module has the form:

`lb 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:

`lb exmpl -r obj3 mod1 mod2`
resulting in `exmpl.lib` containing

`obj1 obj2 mod1 mod2 obj4`

UNIQUENESS

`lb` 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 `lb` to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, option `-u` causes `lb` to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, suppose that the library `exmpl.lib` contains the following:

`obj1 obj2 obj3 obj1 obj3`

The command

```
lb exmpl -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 same name as the module and extension **.o**.

For example, given the library **exmpl.lib** containing the modules

```
obj1 obj2 obj3
```

the command

```
lb exmpl -x
```

extracts all modules from the library, writing **obj1** to **obj1.o**, **obj2** to **obj2.o**, and **obj3** to **obj3.o**.

And the command

```
lb exmpl -x obj2
```

extracts just **obj2** from the library.

THE VERBOSE OPTION

The verbose option, **-v**, causes **lb** to be verbose; that is, to tell you what it is doing.

SILENCE OPTION

The silence option, **-s**, tells **lb** 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:

```
lb exmpl -st > tfil  
lb 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 **lb** 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 **lb**.

DEFINING THE DEFAULT MODULE EXTENSION

Specification of the extension of an object module file is optional; **lb** assures 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**.

```
lb my.lib -e .i mod1 mod2 m
```

HELP

Option **-h** will generate a summary of **lb** functions and options.

NAME	ls - list files and directories
SYNOPSIS	ls [-blprst] [file pattern directory...]
DESCRIPTION	<p>ls lists the names of the files and the contents of the directories whose names are passed to it as arguments. The files you wish to list should be specified by one or more <i>file_pattern</i> arguments. <i>file_pattern</i> may be a <i>filename</i> with or without a path specification, or a <i>filename</i> mixed with wild card characters. The valid wild card characters are:</p> <p>'?' - matches exactly one unknown or variable character.</p> <p>'*' - matches an arbitrary number of unknown characters.</p> <p>Wildcards are used to list multiple file names that are related in some way. For example,</p> <pre>ls *.c</pre> <p>will list all files with a .c extension.</p> <pre>ls ? bc.c</pre> <p>will list filenames such as abc.c, bbc.c, and lbc.c, but not aabc.c</p> <p>If no names are specified, the contents of the current directory are listed.</p> <p>The information is written to the ls standard output device, and thus goes, by default, to the screen. The information can be sent to another device or file by redirecting the ls standard output device in the normal manner.</p> <p>By default, the output is sorted alphabetically in multiple columns with directories displayed in an offsetting color. When directed to a file, the output is always alphabetical, one name to a line.</p>
Options	Meaning
-b	Displays the number of blocks in the file in addition to any other information requested.

- l Generates additional information for each file listed, including attributes, size, and last modification date
- p Forces a blank to be put before each filename and a - before each directory.
- r Reverses the order that has been specified.
- s Causes the list to be sorted by size, with the largest files first.
- t Causes the list to be sorted in chronological order with the most recent files first.

NAME	mkarcv - text file archiver
SYNOPSIS	mkarcv <i>arcfile</i>
DESCRIPTION	mkarcv combines several individual text files into one "archive" file, where the parameter <i>arcfile</i> 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 elsewhere in 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>

NAME	obd - list object code
SYNOPSIS	obd <i>objfile</i> [<i>objfile</i> ,...]
DESCRIPTION	obd lists the loader items in an object file. <i>objfile</i> is the Aztec object module which you desire to examine.

NAME	ord - sort object module list
SYNOPSIS	ord [-v] [infile [outfile]]
DESCRIPTION	<p>ord sorts a list of object filenames for use by the lb 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.</p> <p>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.</p> <p><i>infile</i> 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 <i>infile</i> is not specified, this list is read from the ord standard input. The filenames can be separated by space, tab, or newline characters.</p> <p><i>outfile</i> is the name of the file to which the sorted list is written. If it is not specified, the list is written to the ord standard output. <i>outfile</i> can only be specified if <i>infile</i> is also specified.</p> <p>Option -v causes ord to be verbose, sending messages to its standard error device as it proceeds.</p>

NAME	set - environment variable and exec file utility
SYNOPSIS	set set VAR=<i>string</i>
DESCRIPTION	set is used to examine and set environment variables. Displaying and Setting Environment Variables The first form listed for set causes it to display the name and value of each environment variable. The second form assigns <i>string</i> to the environment variable <i>VAR</i> . In the second form, if <i>string</i> is not given, the specified variable is deleted from the environment.
SEE ALSO	See the Tutorial chapter of your <i>Aztec Shell User Guide</i> for more information on environment variables.

NAME	setdate - set date and time
SYNOPSIS	setdate
DESCRIPTION	<p>setdate allows you to set the date and time. It prompts you as follows:</p> <p>Date (MM/DD/YY) ? Time (HH:MM:SS) ?</p> <p>If you type 02/03, it leaves the year alone. If you type 12 in response to the Time prompt, it assumes 0 for the minutes and seconds.</p>

setdate



Chapter 7 - Debugger

db is a symbolic debugger. It is used to debug programs which have been created using the Aztec C compiler, assembler, and linker.

db has all the standard features of an assembly language debugger. It also has features not found in all debuggers, such as the ability to reference memory locations by name as well as by address, the ability to define sequences of commands to be macros, which can then be activated by entering a single letter, and a flexible mechanism for handling breakpoints.

In addition, **db** has features specifically tailored to its use with Aztec C, such as the ability to list the name and parameters of the currently executing function, and the function that called it, and so on, back to the initial function. Another special feature is the ability to display, on entry and exit from each function, the function's parameters and return value. Finally, **db** supports both scatter-loaded and segmented programs.

Requirements

An Amiga with at least 512k of memory is recommended for use with **db**. The debugger itself uses about 96k.

Preview

This chapter is divided into three sections: **Overview**, which describes **db** features in more detail and introduces the commands; **Using db**, which describes in full detail how to use **db**; and a **Command Summary**.

Overview

db commands consist of one or two characters, the first of which identifies the command category. If there is only one command in the category, then the command has just one letter; otherwise, the command has a second letter which identifies the specific operation to be performed.

BASIC COMMANDS

db has two types of commands for examining memory: display and print, whose first characters are **d** and **p**, respectively. The DISPLAY commands **db** and **dw** simply display hexadecimal bytes and words.

The **PRINT** command, **p**, is more powerful, being able to convert a sequence of one or more possibly different types of data items to ASCII. For example, you can tell the print command that beginning at the location **var** is a sequence of the following items: an **int**, a **float**, and a pointer to a **char** string. The **p** command will convert the two binary items to ASCII and print them, then display the referenced character string.

The **REGISTER** command, **r**, displays and modifies the 68000 registers.

The **MEMORY MODIFY** commands, **m**, modify memory.

The **u** commands **UNASSEMBLE** code; that is, they display it symbolically, in a form similar to its appearance in an assembly language source file.

The **s**,**t** and **g** commands cause your program to be executed. **s** and **t** commands **SINGLE STEP** your program; that is, they execute a specified number of instructions in your program and then return control to **db**. The **t** command differs from the **s** command by single stepping over **jsr** and **bsr** instructions. **g** commands transfer control of the processor unconditionally to your program. In this case, **db** regains control when your program terminates, when an error occurs (such as division by zero), or when a "breakpoint" is taken. Breakpoints are discussed later in this chapter.

? is the **HELP** command: It causes **db** to display a summary of all **db** commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category's commands followed by a ?. For example, typing **m?** gets you information about the memory modification commands (all of whose first letter is **m**).

NAMES

db allows memory locations to be referenced by name as well as by location. It learns a program's global names by reading the symbols from the program file and placing them in a memory-resident symbol table. The linker generates a symbol table hunk for a program in response to the **-w** option.

db only allows global symbols to be accessed by name; automatic variables and static variables cannot be accessed by name.

You can also define names to **db** using the **v** command; The **CLEAR SYMBOLS** command, **cs**, will remove symbols from the memory-resident symbol table.

Code and Data symbols.

db classifies symbols as being either code or data symbols. All symbols in the program's symbol table hunks which occur between the special linker symbols **_H1_org** and **_H1_end** and between **_H2_org** and **_H2_end** are considered to be data symbols, and all others are code symbols.

There are two commands for viewing the symbols which are known to **db**: **dc** and **dd**, which display code and data symbols, respectively.

Operator usage of names.

When a C source program is compiled, all global names are prepended with an underscore character. To refer to symbols within **db**, the name can be typed without the underscore. First, the name will be searched for exactly as typed. If not found, **db** will prepend an underscore and search again.

Note that this means to reference a symbol such as **_main**, you would have to type **_main** to differentiate it from the **main** symbol, since typing **_main** would match before the extra underscore is prepended.

LOADING PROGRAMS AND SYMBOLS

The **db** program is started independently of the program to be debugged. The **al** command causes the debugger to wait till a program is loaded either from the Workbench or from the CLI; the program is stopped before it

executes any instructions. The debugger will also load the symbols from the program at this time if the program is in the current directory. If not in the current directory, the **LOAD SYMBOLS** command, **ls**, can be used.

Unfortunately, the Amiga operating system contains no mechanism for terminating a program from an external process. So, there are two ways for the debugger to terminate the program. The first method is to simply allow the program to resume using the **ar** command and let it terminate normally. If the program is unlikely to terminate normally on its own, the **ak** command checks the symbol table for the **_abort** symbol and then the **exit** symbol and sets the program counter to the first one found, allowing the program to resume.

BREAKPOINTS

Before transferring control of the processor to your program in response to a **g** command, **db** can set **BREAKPOINTS** at specified locations in the code. When your program reaches a breakpoint, **db** regains control.

A breakpoint has a **SKIP COUNT** associated with it, which allows a breakpoint to be passed several times before actually taking the breakpoint and returning control to **db**. When a breakpoint is reached, **db** is always activated; it increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken; that is, **db** retains control of the processor. Otherwise, **db** returns control of the processor to your program after the breakpoint. By default, a breakpoint's skip count is 0; thus, each time the breakpoint is reached, it is taken.

A breakpoint can also have a sequence of **db** commands associated with it. When a breakpoint is taken, these commands will be executed before **db** allows you to enter commands. For example, if you just want to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location, and specify a list of commands to do just that: the first command in the sequence would be a **d** command to display memory, and the second would be a **g** command to continue execution of the program.

There are two ways to define breakpoints: with the **g** command, and with special breakpoint commands, whose first letter is **b**.

The breakpoint commands manipulate a table of breakpoints: there are commands for entering breakpoints into the table, displaying the entries, resetting their counters, and removing them from the table.

There is a difference between a breakpoint defined in a **g** command and those in the breakpoint table: The **g** command breakpoint is temporary, while a table breakpoint is more permanent (it exists until removed from the table). Before transferring control to your program in response to a **g** command, **db** sets all breakpoints that are in the breakpoint table and that are specified in the **g** command itself. When a breakpoint is taken, **db** removes all breakpoints from the code and forgets all about the **g** command breakpoint. The table breakpoints, however, are still in the table and will be set back in memory when control is again returned to your program.

db remembers the skip counter associated with a breakpoint which is in the breakpoint table: When it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (that is, its value in the table); and when a breakpoint is taken, the accumulated count for the breakpoints in memory are saved in the breakpoint table.

MEMORY-CHANGE BREAKPOINTS

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called a **MEMORY-CHANGE BREAKPOINT**, is taken when a specified memory location is changed from or set to a particular value.

With a memory-change breakpoint set, **db** will detect either the function or the instruction which modifies the specified memory location, depending on whether your program was activated using a **g** command or is being single-stepped using an **s** or **t** command, respectively.

When your program is activated with a **g** command and a memory-change breakpoint is set, **db** will examine the specified memory location on entry to, and exit from, each function. It will take a breakpoint, that is, interrupt execution of the program and return control to the operator, when the contents of the memory location meets the specified condition.

When an **s** or **t** command is used to single-step a program and a memory-change breakpoint is set, **db** will examine the specified memory location after each instruction is executed, and take a breakpoint when appropriate.

The **bb**, **bw**, and **bl** commands are used to set and remove memory-change breakpoints.

TRACE MODE

db supports a **TRACE MODE**, which displays information whenever a function is entered or exited.

When entering a function with trace mode enabled, the function or trap name and its arguments are displayed. Optionally, upon exit from a function, a return value is displayed.

The commands **bt** and **bT** affect trace mode: **bt** enables and disables trace mode, and **bT** enables and disables the display of function exit information.

BACKTRACING

When **db** regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location. The **ds** command will display information about the currently executing function, and the function which called it, and so on, back to the Manx function **_main**, which called your function **main**.

ds displays, for each function, its name, arguments which were passed to it, and the address to which it will return.

MACROS

db allows you to define and execute **MACROS**, that is, a sequence of **db** commands.

A macro is associated with a single alphabetical character, so up to 26 macros can be known to **db** at any time.

The **db** command **x** is used both to define and execute a macro.

OTHER FEATURES

Some other features of **db** are:

- The **EVALUATE EXPRESSION** command, **=**.
- The **HELP** command, **?**, which lists commands.
- The **INPUT RADIX** command, **n**, which changes the default radix for input and display.

Using DB

STARTING DB

db is started with a command of the form:

db

or by selecting the **DB** icon and clicking from the Workbench.

When **db** starts, it creates a window with a startup message and then performs several actions. First, if running under V1.2 or later, the window is created without activating it. Next, **db** will look in the current directory for a file with the name **.dbinit**. If found, **db** will open the file, read it, and execute commands from the file.

When **db** reaches the end of the file it checks to see whether it is waiting for a breakpoint or program load and if so skips the next step. If **db** is ready for a command, it will first look in the environment for a variable **DBINIT**. If found, **db** will open the file defined by this variable and read and execute commands from this file as well. In this way, it is possible to do system wide as well as local presets.

When **db** has finished reading either or both files it again looks to see if it is waiting for a breakpoint or a program load. If it is not, then **db** will activate the window and wait for input for the next command. If it is, then **db** will not activate the window.

For example, suppose you enter the following command,

set DBINIT=s:dbinit

then enter the following line in **s:dbinit:**

al

When you start **db**, it will open the window, read **s:dbinit** and wait for a program to load without activating the window. This means you can type:

**db
program**

without having to use the mouse.

TERMINATION

If a program is a process or CLI program, **db** will set a breakpoint on the return address from the program. If this breakpoint is reached, **db** will print a message that the program has terminated normally and will automatically resume the task.

SUPPORT FOR SCATTER-LOADED PROGRAMS

This version of **db** contains support for scatter-loaded programs. Variables which are referenced using absolute addressing are displayed normally. Functions which are accessed through the jump table are displayed in parentheses. However, the names can be used as though not scatter-loaded.

SUPPORT FOR SEGMENTED PROGRAMS (OVERLAYS)

db supports breakpoints in overlays. This is not easy, so there are some caveats. First, any reference to a function in another segment will force the segment into memory. For example, saying:

u over1

where **over1** is a function will force the segment into memory before disassembling.

If breakpoints are set in the permanent breakpoint table, then when the g command is given, any breakpoints in non-resident segments will force the segments into memory before setting the breakpoint. (Think of it as a reference to that symbol.) Note that if a segment is unloaded by the program the breakpoint is lost even if the segment is reloaded by the program. However, if a segment is unloaded and the debugger regains control, the next g command will reload the segment and reset the breakpoint.

One potential problem to watch for: If you set a breakpoint, the program unloads the segment and the function gets called, but the breakpoint is no longer there.

INPUT/OUTPUT

While db is displaying information to the screen, the display can be stopped temporarily by typing ^S. Any key will restart the display. Typing ^C will abort the display.

When db is waiting for a breakpoint or program to load, typing any character will cause db to stop the current program. When db is displaying while running the program (Trace mode, for example) it is best to use ^C, as other characters may be swallowed by the display routines.

COMMANDS

This section describes in detail the db commands. It first defines some terms that are used in the command descriptions.

Definitions

EXPR

An *expr* has the following form:

term [binop *term* ...]

That is, an *expr* can be a single *term* or a series of *terms* separated by binary operators. The binary operators are:

- + addition
- subtraction
- * multiplication

- / division
- % modulus
- & bitwise and
- | bitwise inclusive or
- ^ bitwise exclusive or

All operators have the same precedence, and an unparenthesized *expr* is evaluated left to right. To override the default order of evaluation of an expression, you can parenthesize the relevant parts of the expression.

An *expr* has a 32-bit value. The operators that are applied to the *terms* out of which the *expr* is built affect just this 32-bit value.

TERM

A *term* always resolves to a numeric value, and can be one of the following:

<i>register</i>	<i>constant</i>
<i>-term</i>	<i>addr</i>
<i>*addr</i>	<i>#addr</i>
	<i>(expr)</i>

These names are defined in the following paragraphs.

- **REGISTERS** are specified by their standard names; **a0**, **d0**, **pc** and so on. The value of the *term* is the contents of the register.
- **CONSTANT** is a decimal, hexadecimal, octal number, or a character.

A sequence of digits preceded by **0x** is taken to be a hexadecimal number and those preceded by **0b** are binary numbers. A sequence of digits with a leading **0o** is taken to be an octal value. Digit strings ending in **.** are taken to be decimal values. If none of these prefixes or suffixes are present, the radix of the value is taken from the current radix. The default radix is hexadecimal. Note that if the current radix is set to hexadecimal, numbers must start with a digit to distinguish them from symbols (i.e. **fabc** will be taken to be a symbol where **0fabc** will be taken to be a hexadecimal number.) However, if a symbol is not matched, and all the letters are hex digits, the sym-

bol will be treated as a number. This is useful, but beware of something like:

```
dw foo+a0
```

because **a0** will use the value in register **a0** and not the value **0xa0**.

A character is represented by the character, surrounded by single quotes, as in 'x'. The value of a character constant is its ASCII value.

Certain characters, the single quote ' and the backslash \ may also be defined within the single quotes. These are identified by a leading backslash character, and are:

char	hex value	db notation
newline	0a	\n
horizontal tab	09	\t
backspace	08	\b
carriage return	0d	\r
form feed	0c	\f
backslash	5c	\\\
single quote	27	\'
bit pattern	ddd	\ddd

- **ADDR and *ADDR** When a *term* consists of a * followed by an *addr*, the value of the *term* is the contents of the 32-bit field referred to by the *addr*. For example:

- ***var** The contents of the *var* field;
- ***a3** The contents of the 32-bit field in the data segment pointed at by **a3**;
- ***sp** The contents of the 32-bit field on the top of the stack;
- *(**lbl+2**) The contents of the 32-bit field referred to by **lbl+2**;

Because an *addr* can itself be an *expr*, the **addr* term may require extra parentheses. For example:

***sp+2**

is equivalent to ***(sp+2)** and not **(*sp)+2**. The value of the first interpretation is the contents of the second word on the stack, while the value of the second is two plus the contents of the first word on the stack.

- **PERIOD (.)** The value of a *term* consisting of a period, ".", is the starting address *addr* of the last similar command. For example, if ten bytes of memory were displayed using the **db** command, as in:

db 0x100,10

then "." would be set to 0x100 for the next **db** or **dw** command. If the next **db** or **dw** command is

dw .

the same 10 bytes would be displayed as words.

"." has a separate value for the **u** command, for the **db**, **dw**, and **m** commands, and for the **p** command. An **m** command never modifies its associated "..".

- @ [*function*] symbol has as its value the return address of the specified function. The function name is optional, and defaults to the current function. The main use for @ is in the **g** command. For example:

g @

transfers control to your program, and sets a breakpoint at the return address of the current function.

As another example:

```
g @putc
```

transfers control to your program. When the function `putc` is reached, a breakpoint will be set at the address to which it will return.

ADDR

An *addr* defines the address of a location in memory, and has the form:

expr

Here are some examples of *addr*:

```
pc  
main+10  
. -40  
*sp+8  
data+*(a6+6)
```

RANGE

A *range* defines a block of memory. It has one of the following forms:

```
addr,cnt  
addr>addr  
addr  
,cnt
```

The form *addr,cnt* specifies the starting address, *addr*, and a number, *cnt*. *cnt* is interpreted differently by different commands. For example, the DISASSEMBLE CODE command, `u`, will display *cnt* lines, while the DISPLAY BYTES command, `db`, will display *cnt* bytes.

The form *addr>addr* specifies the starting and ending addresses of *range*.

A full *range* need not be explicitly specified, because **db** remembers the last-used *range* and will set unspecified *range* parameters from the remembered values:

- When a *range* is specified which consists of a single *addr*, the last used *cnt* is used.
- When a *range* is specified which consists of *cnt*, the next consecutive address is used, and the remembered *cnt* is changed to the new value.
- When nothing is specified as the *range*, the next consecutive address is used as the starting *addr*, and the *cnt* is set to the remembered value.

CMDLIST

A *cmdlist* is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

COMMAND [; COMMAND . . .]

If a macro is in a *cmdlist*, it must be the last command in the list.

COMMAND DESCRIPTIONS

The following descriptions of debugger commands use terms and concepts which were presented in the preceding sections.

The commands are listed alphabetically. For an index, see the command summary which follows the descriptions.

The Amiga Commands

► add	display device list
► adi	display interrupt list
► adl	display library list
► adp	display port list
► adr	display resource list

Syntax:

add
adi
adl
adp
adr

Description:

These commands play the addresses and names of one of the various lists pointed to by **ExecBase** in the Amiga.

- **ai** display task information

Syntax:

ai

Description:

This command displays information about a particular task. If the task is a process or has been run from the CLI, additional information besides the task information is displayed.

If a task has already been selected, the **ai** command displays the information for the selected task. Otherwise, a list of tasks is displayed and the debugger prompts for you to enter a task.

- **ak** kill the current task

Syntax:

ak

Description:

This command is used to terminate the program or task currently being debugged. It does this by transferring control to the `_abort()` or `exit()` function of the program.

The Amiga operating system has no provisions for tracking memory or resource allocation. As a result, it is not possible to "kill" an executing task from an external one. Thus, if the program being debugged is unable to resume and exit normally, the system must be rebooted.

The only alternative is to force the program into its `exit()` function. This may or may not work depending upon the state of the program when the `ak` command is given. At the least, some memory is likely to be lost.

This command is made available as a shortcut for the equivalent command `r pc=exit`, and discretion in its use is advised.

- **a1** debug the next program or task
- **aL** debug the next task without symbols

Syntax:

a1
aL

Description:

Since the debugger operates as an independent task, it cannot load programs directly. Instead, it replaces certain vectors in the system and waits for the next program or task to be loaded. The debugger takes control before the task or program begins to execute.

Thus, to debug a program, this command is given and the debugger waits while you switch to a new window and either type a CLI command or click on a Workbench icon to start the program.

After the program is loaded and the debugger gains control, the symbols are loaded from the program file unless the `aL` command is used. Under 1.2, the window is automatically activated when the program is loaded.

- **an** create a new debugging window
- **aN** create a new debugging window with new symbols

Syntax:

an
aN

Description:

When the debugger is started, it creates its own window for commands and displays. With the **an** command it is possible to create additional windows for debugging more than one task at a time. If the **an** command is used, the new window will share symbols with the window active when the command was given. This is useful when debugging a sub-task that is a part of the active program. Note that the **al** command will not read symbols if a symbol table already exists.

When the **aN** command is used, the new window is created with an independent symbol table. This is useful for debugging several programs simultaneously.

- **am** memory information

Syntax:

am

Description:

This command displays the amount of free memory in the system.

- **ap** debug a crashed program (post-mortem)
- **aP** debug a crashed program without symbols

Syntax:

ap
aP

Description:

When a program run from the CLI or Workbench crashes because of an address or bus error or a divide by zero, etc., the trap is handled by a special DOS handler that puts up the "Software Failure" requester. Selecting RETRY has no effect, and selecting CANCEL passes control to exec's trap handler which displays the **guru**. This function provides a way to examine the environment after the crash.

If **db** is running, or if you can start up **db** from another window, then using the **ap** command, **db** can act as though it was in control the whole time. When you give the command, **db** will ask you to select the task to be debugged. It will also make an informed guess as to which task it is. After selecting the task, **db** will instruct you to select the CANCEL button in the requester.

db will have control right at the point of error. At this point, it is possible to examine the state of things to determine why it happened and possibly to even recover by patching or skipping the code in error. Note that this is not always possible, since memory may have been corrupted before the error occurred.

Note also that if multiple requesters are present for different tasks, this will only work if you choose the most recent task that failed and click the proper requester.

► **aq** close all windows

Syntax:

aq

Description:

When more than one window is created using the **an** command, individual windows can be closed using the **q** command. The **aq** command will close all the windows and terminate the debugger. When the last window is closed, the debugger terminates.

When a window is closed, if a task was stopped for debugging, the task is allowed to resume.

The **aq** command performs a **q** command for each of the windows that is open.

- **ar** allow the current task to resume

Syntax:

ar

Description:

When a task has been selected for debugging, it is kept in a stopped state while commands are being given. The **ar** command deselects the task and allows **db** to resume. The window returns to the state it was in before a task was selected.

- **as** select a task to debug
- **aS** select a task without symbols

Syntax:

as
aS

Description:

It is possible for the debugger to debug a task or program that is already running in the system. The **as** command displays a list of all tasks in the system and prompts for the number of the task to debug. That task is stopped and the symbols for that task are loaded from the program file unless the **aS** form of the command is used.

- **at** display all tasks

Syntax:

at

Description:

The **at** command displays a list of all tasks in the system along with their priority, the address of the task block, the status and the name.

The Breakpoint Commands

- **bb** set byte memory-change breakpoint
- **bw** set word memory-change breakpoint
- **bl** set long memory-change breakpoint

Syntax:

```
bb  
bw  
bl  
bb  addr [==[val]]  
bb  addr [!= [val]]  
bw  addr [==[val]]  
bw  addr [!= [val]]  
bl  addr [==[val]]  
bl  addr [!= [val]]
```

Description:

These commands are used to set and clear a memory-change breakpoint, with the parameterized versions used to set breakpoints and the parameterless versions to clear them. The **bb** command is used to monitor a one-byte field, the **bw** command to monitor a two-byte (word) field and the **bl** command to monitor a four-byte field.

In the parameterized form of the commands, *addr* specifies the field to be monitored.

With the **==** form, the breakpoint will be triggered when the debugger detects that the field is equal to the specified value, *val*.

With the **!=** form, the breakpoint will be triggered when the debugger detects that the field is different from the specified value.

The *val* parameter is optional. If not specified, **db** defaults to the current value at the *addr*. If the comparison field is also not specified, the default is !=. Thus, to break when a location is changed, use the command:

bw addr

- **bc** clear a single breakpoint
- **bC** clear all breakpoints

Syntax:

bc addr
bC

Description:

These commands delete breakpoints from the breakpoint table.

bc deletes the single breakpoint specified by the address *addr*, and **bC** deletes all breakpoints from the table.

- **bd** display breakpoints

Syntax:

bd

Description:

bd displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

- Its address, using a symbolic name, if possible.
- The number of times reached without breakpointing.
- The skip count;
- The command list, if any.

For example, a **bd** display might be:

address	hits	skip	command
<code>_printf</code>	1	2	
<code>_putc</code>	0	0	<code>db _Cbuffs</code>

In this example, two breakpoints are in the table. The first is at the beginning of the function `_printf`; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the second time `_printf` is reached.

The second breakpoint is at the function `_putc`; a breakpoint will be taken each time the function is reached, and will display memory, in bytes, starting at `_Cbuffs`.

► **bh** set memory hash breakpoint

Syntax:

`bh [range]`

Description:

bh performs a simple checksum on the specified range of memory. Breakpoints are automatically set at the entry and exit of all functions. The checksum is checked each time the debugger regains control of the task. If the checksum is ever detected to be different, the debugger will stop the program at that point.

► **bq** toggle low memory checksum

Syntax:

`bq`

Description:

When the debugger starts, it checksums the first 256 bytes of memory. Each time the debugger regains control, this checksum is calculated. If the

checksum differs from the saved value, a message is displayed and the program stopped. The new checksum is saved so multiple breakpoints are not generated.

The **bq** command toggles whether the checksum is done or not. Turning off checksumming will speed single-stepping since otherwise the checksum is computed for each instruction executed.

- **br** reset breakpoint counters

Syntax:

br [*addr*]

Description:

br resets the "hit" counter for the specified breakpoint which is at the address, *addr*. If *addr* is not given, the "hit" counters for all breakpoints in the breakpoint table are reset.

- **bs** set or modify a breakpoint

Syntax:

[#] **bs** *addr* [*cmdlist*]

Description:

bs enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter **#** is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached it will be taken.

The optional parameter *cmdlist* is a list of debugger commands to be executed when the breakpoint is taken.

- **bt** toggle the trace mode flag
- **bT** toggle the return trace mode flag

Syntax:

bt
bT

Description:

bt and **bT** toggle the trace mode and return trace mode flags, respectively.

The state of the trace mode flag determines whether trace mode is enabled or disabled.

The state of the return trace mode flag determines whether the tracing of a function's return is enabled or disabled. If trace mode is disabled, the return trace mode flag has no effect.

Note that recursive calls will not handle the return trace mode correctly.

- **bu** user defined breakpoint

Syntax:

bu *addr*

Description:

bu sets the address of a user-defined breakpoint function.

This command sets a pointer to a function which is called each time the debugger regains control. The function has no effect when it returns a zero value. When a non-zero value is returned, a message displaying the value is displayed and the task stopped.

The function is called as a C function and is expected to follow the standard C register saving conventions. Passed as an argument to the function is the address of an array of longs which contain the tasks register values, includ-

ing the program counter and status register. The order of items in the array is:

registers d0-d7
registers a0-a7
status register
program counter

This function can be used to breakpoint on special conditions. For example, the program can be stopped when the d0 and d1 registers contain the same value. The function would look like this:

```
moveq    #0,d0
move.l   4(sp),a0
move.l   (a0),d1
cmp      4(a0),d1
bne     1$
moveq    #1,d0
1$      rts
```

Setting the address to zero disables the breakpoint.

The Clear Commands

► cs clear symbol table

Syntax:

cs

Description:

cs removes all symbols from the debugger's memory-resident symbol table.

The Display Commands

- **db** display memory in bytes
- **dw** display memory in words
- **dl** display memory as longs
- **d** display memory in last format

Syntax:

```
db [range]
dw [range]
dl [range]
d [range]
```

Description:

The **db**, **dw** and **dl** commands display successive bytes, words and double words of memory, respectively. **d** displays memory using the last format specified; for example, if **d** is entered, and **db** was the last DISPLAY MEMORY command, then **d** will display bytes, too.

The starting address of the *range* parameter is optional; if not specified, it defaults to the ending address of the last display's *range*, plus one.

Each line of the display begins with the address, followed by a hexadecimal display of 16 bytes, 8 words or 4 double words, followed by an ASCII display, by bytes, of the same data. For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

- **dc** display all code symbols

Syntax:

```
dc
```

Description:

dc lists all the code symbols in the memory-resident symbol table and all user-defined symbols.

For each symbol, name and address are displayed.

- **dd** display all data symbols

Syntax:

dd

Description:

dd lists all the data symbols in the memory-resident symbol table.

For each symbol, name and address are displayed.

- **dg** display global values

Syntax:

dg

Description:

For each data symbol in the debugger's symbol table, **dg** displays the contents of the 16-bit field referenced by that symbol.

- **ds** display stack backtrace

Syntax:

ds

Description:

ds displays information about the current function, the function which called it, and so on, back to `_main()`, the Manx function which called your function `main()`.

For each function, the information consists of the function's name, the parameters passed to it, and the address to which it will return. The arguments are displayed as a series of 16-bit hex values. If an argument is actually of type long or double, it will be displayed as separate words.

ds determines the number of parameters by looking at the instructions which follow the address to which the function will return. **ds** assumes that the **a5** register points to the C stack frame for the current function, unless the current instruction is within 4 bytes of the start of the function.

The Go commands

- **g** execute the program
- **G** execute the program, without setting table breakpoints

Syntax:

```
[#]g [@function] [addr] [;cmdlist]  
[#]G [@function] [addr] [;cmdlist]
```

Description:

The **g** commands transfer control of the processor to your program, at the address specified by **pc**. Your program then executes until it terminates, an error such as division by zero occurs, or a breakpoint is taken; control then returns to the debugger program.

The parameters to the **g** commands allow one or two temporary breakpoints to be set in memory before your program is executed.

The difference between the **g** and the **G** command is that the **G** command sets in memory just the breakpoints specified in the command itself, while the **g** command also sets the breakpoints specified in the breakpoint table.

The **#** and **addr** parameters define one of the temporary breakpoints that a Go command can set:

- **#** is the skip count for the breakpoint; it defaults to zero, meaning that the breakpoint is taken every time it is reached.
- **addr** is the address for the breakpoint;

The *@ function* parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function is not specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function will return. In this case, the breakpoint is not set until the function is entered; thus, in programs which call the function from several different places, the breakpoint will be set at the actual address to which the function will return.

The *; cmdlist* parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to execute once a breakpoint which is specified in the *go* command is taken. If this parameter is not specified, it defaults to the command list used for the last temporary breakpoint.

Before setting breakpoints and transferring control to your program, the debugger single-steps your program, (that is, causes it to execute one instruction). This allows you to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

The Load Commands

- **ls** load symbols

Syntax:

ls progfile

Description:

ls loads symbols from the specified program file into the debugger's memory-resident symbol table, after first clearing the memory-resident table of all but those symbols defined with the *v* command.

The Memory Modification Commands

- **ma** allocate some memory

Syntax:

ma val

Description:

ma allocates *val* bytes of memory and displays the address of the memory.

This command is mostly used to allocate memory for patches and possibly to hand assemble a small user break routine. The “.” associated with memory change and display is set to the address as well making it possible to load patches or user functions from a file.

- **mb** modify bytes of memory
- **mw** modify words of memory
- **ml** modify double words of memory

Syntax:

```
mb  addr expr1[expr2 ...]  
mw  addr expr1 [expr2 ...]  
ml  addr expr1 [expr2 ...]
```

Description:

mb, **mw** and **ml** modify bytes and words of memory, respectively.

The parameter *addr* specifies the address of the first byte or word to be modified.

The *expr* parameters are expressions, whose resulting values are set in memory, with *expr1* set in the first byte or word specified, *expr2* set in the next higher byte or word, and so on.

The *expr* parameters can be separated by spaces or commas.

- **mc** compare memory

Syntax:

```
mc  range = addr
```

Description:

mc compares two blocks of memory and, for each comparison which fails, displays the corresponding address and value.

range specifies one of the blocks of memory. The second begins at *addr* and has the same length as the first block.

- **mf** fill memory

Syntax:

mf *range = expr*

Description:

mf sets each byte in a block of memory to a specified value.

The *range* parameter specifies the memory block, and *expr* an expression whose resulting value is the value to be set in the range.

- **mm** move memory

Syntax:

mm *range = addr*

Description:

mm copies one block of memory to another.

The *range* parameter specifies the source block and *addr* the starting address of the block to be modified.

- **ms** search memory

Syntax:

ms *range = expr1 [expr2 ...]*

Description:

ms searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

range specifies the block of memory. The *expr* parameters are expressions, each of whose resulting values is one byte of the search sequence.

The Radix Command

- **n** change radix

Syntax:

nx
n

Description:

This command changes the default radix (hexadecimal) for user input or debugger display. The suffix *x* is a single character **b**, **d**, **o**, or **x** which changes the default radix to binary, decimal, octal, or hexadecimal respectively. If suffix *x* is omitted, a message giving the current radix is displayed.

The radix can be forced to a given value by specifying one of the following prefixes before the desired number:

0x	hex
0o	octal
0b	binary

To display a number in decimal, the number must have a . (period) appended to it.

The "Print" Command

- **p** formatted print

Syntax:

p[format] [addr][,count]

Description:

p displays a formatted section of memory. Data items are converted to a displayable form as directed by the format conversion string *format*.

format is a list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

p works its way through the *format* string, converting and displaying data in memory as requested by the format string items. When **p** reaches an item in the *format* string, it converts the data at its current address as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed, and processes the next data item as directed by the next format string item.

The *format* string is optional; if not specified, the *format* string used by the previous **p** command is used.

addr specifies the address of the first data item that **p** is to convert and display. If *addr* is not entered, the starting address is assumed to be the print command's current address. Normally, this is the address of the first byte beyond the last data item converted by the last **p** command. However, there is a *format* item that causes **p** to remember the address contained in the current data item, and then make that the current address after it finishes processing the entire format string.

count specifies the number of times that **p** is to work its way through the *format* string. Each time through, **p** begins at the current address that was left by the last time through. If *count* is not specified, it defaults to one time.

The format items have the form:

[rpt][indir_flg][size]desc_code

where

- *desc_code* is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code **d** converts the two-byte binary value at the current address to decimal, and prints it. So if **var** is an **int**, the following would print its value in decimal:

pd var

The code **x** says "take the two-byte binary value at the current address, convert it to hexadecimal, and print the result". So the hexadecimal value of **var** could be printed with the command:

```
px var
```

- *indir_flg* is a string of zero or more * characters, which are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code*.

To find the data object corresponding to a format item that has indirection indicators, **p** begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address. At the end of this process, the **p** command's current address is simply incremented past the first pointer.

A * specifies that the pointer within the field referenced by the current data object address is four bytes long. This pointer is the offset component of the new data object address from the last segment referenced.

For example, if the variable **cp** is a pointer to a character string (that is, its declaration is **char *cp**), then the string pointed at by **cp** could be printed by the command

```
p*s cp
```

Here we have made use of the **s desc_code**, which specifies that the data object is a character string, and that the string's characters are to be printed, with possible modifications as noted below, up to a terminating null character. After this command, the **p** command's current address is set to the byte immediately following **cp**.

As another example, if **cpp** is a pointer to an array of pointers to character strings (that is, the declaration of **cpp** is **char **cpp**), then the string pointed at by the first element of the array could be displayed with the command

```
p**s cpp
```

Following this command, the **p** command's current address is set to the byte following **cpp**.

- The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count.

For example, if **a** is an array of **floats**, then the first five items in this array could be displayed with the command

```
p5f a
```

This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is **f**. This command is equivalent to the command **pfffff a**. At the end of this command, the **p** command's current address is set to the address of the byte following the last displayed **float**.

- The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item does not use indirection, *size* has the same effect as *rpt*; for example, in the **p5f a** command above, the 5 could be interpreted as being a *size* parameter instead of a *rpt* parameter.

When the format item uses indirection, the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module defines **lpp** as a pointer to an array of pointers to an array of **longs** (the declaration of **lpp** is **long **ip**), then the following command will display the first four **longs** pointed at by the first element of the pointer array:

```
p**4D lpp
```

Here we have used the **D** *desc_code*, which specifies that a four-byte signed binary value is to be converted to decimal and printed. The following command would display the first three longs pointed at by the first three elements of the pointer array:

p3*4D *1pp

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items **4*d** and ***4d**. The first causes the print command to take the item at the current address as a pointer, increment the current address by four, convert to decimal and print the two-byte value referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by sixteen.

The second item causes the print command to again take the item at the current address as a pointer, increment the current address by four, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has been advanced by four.

As an examplee of the use of format strings containing several format items, consider the following code in a program that uses short data pointers:

```
struct {
    int *ip;
    float flt;
    char *cp;
} var = {&i, 3.14159, "ralph"};
int i=2;
```

The command

p*d2-xf*s2-x var

will print

2xxxx 3.14159 ralph yyyy

where *xxxx* is the hexadecimal address of *i* and *yyyy* is the hexadecimal address of the string.

A complete list of desc_codes

- b** Convert to hexadecimal and print a byte.
- d** Convert to hexadecimal and print a 2-byte signed binary value.
- D** Convert to decimal and print a 4-byte signed binary value.
- f** Convert and print a four-byte float.
- F** Convert and print an eight-byte double.
- o** Convert an octal and print a two-byte field.
- O** Convert to octal and print a four-byte field.
- x** Convert to hexadecimal and print a two-byte field.
- X** Convert to hexadecimal and print a four-byte field.
- u** Convert to decimal and print an unsigned, two-byte value.
- U** Convert to decimal and print an unsigned four-byte value.
- p** Print a pointer in address form with translation.
- P** Print a pointer in address form without translation.
- c** Print a character with translation
- C** Print a character without translation.
- s** Print a string up to a terminating null byte with translation.
- S** Print a string up to a terminating null byte without translation.

For the C and S *desc_codes* each character is printed as is , with no translations.

For the c and s codes, printable ASCII characters (that is, whose hex value is between 0x20 and 0x7f) are printed as is. A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, 0x41, 0x1, and 0x81 would be printed as A, ^A, and ' ^A, respectively.

The following *desc_codes* can be used to assist in the formatting of the p output:

character	output
N or n	Output a newline character
R or r	Output a blank character
T or t	Output a tab character
"string"	Output "string"

These characters can be preceded by a count specifying number of characters or strings to be output.

The next group of *desc_codes* change the p command's notion of the current address. They do not cause any printing.

- ^ Back up the current address by the size of the last data item.
- or + Back up or advance, respectively, the current address by size bytes, where size is a decimal value preceding the code. If size is not specified, it defaults to one byte.
- A or a Remember the pointer that is contained in the current data object: If this pointer is not null, set the p command's current address to this value after the entire format string has been processed.

If the pointer is null, set the p command's current address to the value it had before the entire format string was processed.

The A and a *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines the structure for a symbol table item and declares **sym_head** to be a pointer to this structure. The program that uses this structure and field will chain symbol table items together and set a pointer to the head of the chain in **sym_head**.

```
struct symbol {
    struct symbol * sym_next;
    char *sym_name;
    unsigned sym_val;
} *sym_head;
```

The following command would display the symbol table item pointed at by **sym_head** and then set the **p** command's current address to the next symbol table item, which is pointed at by the **sym_next** field in the first item:

```
PA"symbol name=*snt"value=x sym_head
```

After this command is entered, you can display successive symbol table items by simply entering:

P

The **p** command's current address is correctly set to the next table item, and since a format string is not specified, the **p** command will use the one that it last used.

You can print out multiple symbol table items by entering a single **p** command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command will follow the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose **sym_next** pointer is null. In the latter case, it will terminate and leave the **p** command's current address set to the address of the last symbol table item. For example, entering:

```
PA"symbol name=*snt"value=x sym_head,100
```

will print symbol table items until it either prints 100 items or it prints an item having a null **sym_next** pointer.

The Quit command

► q quit the debugger

Syntax:

q

Description:

When more than one window is created using the **an** command, individual windows can be closed using the **q** command. When the last window is closed, the debugger terminates.

When a window is closed, if a task was stopped for debugging, the task is allowed to resume.

The Register command

- **r** register display

Syntax:

r

Description:

r displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers. If a 68881 is in use for this task, its registers are displayed as well, though they may not be modified.

The parameterized version modifies the contents of a register, with **<reg>** being the name of the register to be modified, and **expr** an expression whose resulting value is to be set into the register.

The Single Step commands

- **s** single step with display
► **S** single step without display
► **t** single step with display over subroutines
► **T** single step without display over subroutines

Syntax:

[#] **s** [*cmdlist*]
[#] **S** [*cmdlist*]
[#] **t** [*cmdlist*]
[#] **T** [*cmdlist*]

Description:

These commands **SINGLE STEP** your program; that is, they execute its instructions one by one. The **s** versions of the command single-step through each and every instruction. The **t** versions allow you to treat **jsr** and **bsr** instructions specially. The debugger does not gain control until the routine called by the instruction returns.

The optional **#** parameter specifies the number of instructions to be executed; it defaults to one instruction. However, a value of **0** specifies a continuous mode which will only stop by user intervention or when one of the other breakpoints is encountered.

The optional *cmdlist* parameter is a list of debugger commands to be executed after each single-step.

The commands differ in that **s** and **t** display information after each single-step, whereas **S** and **T** only display information after the last single-step.

The displayed information consists of the registers and a disassembly of the next instruction to be executed.

The Unassemble commands

- **u** unassemble memory, with symbols
- **U** unassemble memory, without symbols

Syntax:

u range
U range

Description:

These commands **DISASSEMBLE** a range of memory; that is, they display the assembly language instructions in the range.

The **u** and **U** commands differ in that the **u** command will make use of the symbol table during disassembly and the **U** command will not. Also, the **U** command displays, for each instruction, the hex value of each byte of the instruction, whereas the **u** command will not.

With the **u** command, the disassembly of an instruction which references memory displays the location as the symbol nearest to the location plus an offset, if possible. With the **U** command, the location is displayed as a hexadecimal value.

The *range* parameter specifies the area of memory to be disassembled. It gives the starting address, and either the number of instructions to be disassembled, or the ending address of the area.

The Variable commands

- **v** create a new symbol
- **V** modify the value of an existing symbol

Syntax:

```
v      symbol = addr  
V      symbol = addr
```

Description:

The **v** and **V** commands are used to create a new symbol or modify the value for an existing symbol, respectively, in the debugger's memory resident symbol table.

symbol is the name of the symbol being created or modified, and *addr* is its address.

The symbol will be classified as a code symbol.

The Macro command

- **x** macro command

Syntax:

```
xc  
xc = cmdlist  
x?
```

Description:

The **x** command defines or executes a sequence of debugger commands, called a **MACRO**. It can also list the defined macros.

A macro is associated with any letter, so up to 26 macros can be defined. Case is insignificant. A macro is defined by typing the letter **x**, followed by the letter with which the macro is to be associated. Then follows an **=** character and the macro's list of debugger commands, with the commands separated by semicolons. To execute a macro, type **x**, followed by the letter with which the macro is associated, followed by a carriage return.

Macros can be listed with the command **x?**. The output format is suitable for reading back in from a file.

The Display Expression Command

- = display the value of an expression

Syntax:

= *expr*

Description:

This command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a symbol table has been loaded, the closest symbol is displayed as well.

The Redirect Input/Output Commands

- < take input from file
► > log output to file
► >> log commands only

Syntax:

<*file*
>*file*
>>*file*

Description:

These commands are used to temporarily redirect input and output to the files specified.

The < command causes the input which normally comes from the keyboard to be taken from the named file instead. This continues until the end of file is reached. This is especially useful for defining macros.

The > command causes the output of all commands to be displayed in the window and also written to the file specified. This continues until either a different file is specified using the > command or until the > command is used with no file name. In this case, the current file is closed and redirection stops.

The >> command causes a log of the commands only to be saved in a file. This is useful if a complicated set of commands is needed to get a program to a certain point for debugging. After the commands have been logged to the file, the file can be used as input via the < command to retrace the steps to reach the same point.

The Help command

► ? list commands

Syntax:

[*letter*]?

Description:

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the letter, the ?, and return. For example, the listing for the display commands is d?; thus you can type d?, followed by return to get a listing of all the display commands.

Command Summary

AMIGA COMMANDS

add	display device list
adi	display interrupt list
adl	display library list
adp	display port list
adr	display resource list
ai	display task information
ak	kill the current task
al/aL	wait for next task load with/without symbols
am	display memory information
an	make a new debug window
ap/aP	select task for post-mortem
aq	close all windows
ar	release current task
as/aS	select task to stop with/ without symbols
at	display task list

BREAKPOINT COMMANDS

bb/bw/bl	set byte/word/long memory-change breakpoint
bc/bC	clear one/all breakpoints
bd	display the breakpoint table
bh	set checksum breakpoint
bq	toggle low memory checksum
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode
bu	set user defined breakpoint

CLEAR COMMANDS

cs clear all symbols

DISPLAY COMMANDS

db/dw/dl/d	display memory in bytes/words/longs/last format
dc/dd	display code/data symbols
dg	display global values
ds	display stack backtrace

GO COMMANDS

g/G execute user's program

LOAD COMMANDS

ls load symbol

MEMORY MODIFICATION COMMANDS

ma	allocate some memory
mb/mw/ml	modify bytes/words/longs of memory
mc	compare areas of memory
mf	fill memory
mm	move memory
ms	search memory

RADIX COMMAND

n change the default radix for input and display

FORMATTED PRINT COMMANDS

p generate formatted print

QUIT COMMAND

q close current window

REGISTER COMMAND

r register display

SINGLE STEP COMMANDS

s/S single step with/without display
t/T single step with/without display
over calls

UNASSEMBLY COMMANDS

u/U assemble memory

VARIABLE COMMAND

v/V create/modify symbol

MACRO COMMAND

x define or modify a command macro

DISPLAY EXPRESSION

= display value of an expression

INPUT/OUTPUT COMMANDS

< take input from file
> log output to file
>> log commands to file

HELP COMMAND

? list debugger commands

TECHNICAL INFORMATION

8

Chapter 8 - Technical Information

This chapter discusses technical topics and topics that could not be conveniently discussed elsewhere.

It is divided into the following sections:

1. **Program Organization.** Discusses the factors that affect the memory organization of a program.
2. **Segmentation and Segmented Code.** Describes the partitioning of a program's executable code into several segments.
3. **Object Module Libraries.** Discusses the object module libraries provided with Aztec C.
4. **Assembly Language Functions.** Describes how to interface assembly language routines with C routines.
5. **Interrupt Handlers.** Describes routines to provide real-time response to events.
6. **Creating Subtasks.** Discusses programmer use of multi-tasking supported by Amiga.
7. **Floating Point Information.**

8. Building Programs that Run from the Workbench. Describes how to build programs that run from the Workbench.

9. Creating Detachable Programs.

Note: In this chapter, the number zero is represented by the character "Ø", in order to better distinguish between it and the letter "O".

Program Organization

Programs created with the Aztec C system are automatically split into several different sections by the compiler and linker. These sections are called segments. Code is split into these distinct segments so that the Amiga can most efficiently utilize its memory, because segments need not be contiguous, or next to each other, within the Amiga's memory space. The following segments are used for a program:

- one or more code segments
- an initialized data segment
- an uninitialized data segment
- a stack segment
- zero or more heap segments

The code segment(s) contain all of the program's executable code. You can specify how many code segments there will be, (up to 256) and what routines will be contained within each one. See below for more details.

The initialized data segment contains all of the program's initialized data. Initialized data is data which is initialized at the time of its declaration. For example, the following code fragment:

```
int i = 5;  
main ()  
{  
...  
}
```

declares **i** as an initialized data object. Note that only global variables and static variables which are initialized count as initialized data; normal local variables are always placed on the stack. Initialized data is set to a value at compile time, and thus takes up space in the executable file.

The uninitialized data segment is used for all uninitialized global and static variables. In the following code fragment, **i** is uninitialized data because it is given its initial value from within a function, and not where it is declared:

```
int i;
main ()
{
    i = 5;
}
```

Initialized data does not take up space in the executable file, but it must be initialized at runtime from within a function before it is used.

The stack segment is used for local (also known as automatic) variables, as well as for function call address information.

The heap segment(s) are used by the standard memory allocation routines (**malloc()**, **calloc()**, etc.), and indirectly by the standard I/O routines, to allocate memory at runtime. These segments do not take up any space within the program file.

PLACEMENT OF SEGMENTS OF MEMORY

The locations of the code, initialized and uninitialized data segments, and stack are set by the Amiga loader. The placement in memory of any particular program segment is independent of the placement of any other segments, with one exception: If any of the program's modules use the small data memory model, the program's initialized and uninitialized data segments occupy a single, contiguous block of memory, with the uninitialized data following the initialized data.

Linker options **+c** and **+f** can be used to force a program's code, initialized data, and/or uninitialized data segments to be loaded into chip or fast memory. For details, see the **Linker** chapter. A common application for this

is to force all initialized data into chip memory, so that the Amiga custom graphics chips can access this data.

SEGMENT SIZE

Due to various factors, certain segments may have size restrictions placed upon them. The exact restrictions depend on the segment type:

- **Code segments:** There is no limit on the size of a program's code segments, regardless of whether the small code or large code memory models are used.
- **Initialized and uninitialized data segments:** If all of a program's modules use the large data memory model, there is no limit to the sizes of its initialized and uninitialized data segments. If any of its modules uses the small data memory model, then register **a4** points 32K from the beginning of the program's data area, and data that small data modules attempt to access must lie within 32k bytes on either side of the location pointed at by **a4**. Because of this limitation, it is strongly recommended that you do not mix small and large data model files, unless you carefully plan how your files will be linked.
- **Stack segment:** When a program is loaded within the CLI environment, the size of its stack area is set to the value specified in the most recently-entered stack command, or to the CLI default value if a stack command has not been entered. When a program is loaded by the Workbench, the program's **.info** files define the size of its stack area.
- **Heap segment(s):** The size of a program's heap segments depends on its use of the dynamic allocation routines and standard I/O routines. If none of these routines are used, it is possible to effectively have no heap segment. For a discussion of the size of dynamically-allocated buffers which are placed in the heap segments, see the **Library Overview** chapter in your *Aztec C Library Manual*.

Segmentation and Segmented Code

The following two sections describe how the executable code portion of a program can be split into multiple segments to allow a large program to run within a limited amount of memory. The section entitled **SEGMENTATION** discusses how segmentation works, and the **SEGMENTED CODE** section describes how to create segmented code.

SEGMENTATION

Code segmentation is very similar to overlay systems which are available on other computer systems. The primary difference between the Amiga's segmentation scheme and other overlay techniques is that your source code does not need to be modified to take advantage of it. All segmentation setup is done at link time.

When segmentation is used, a program should be designed so that all major functions are mutually exclusive of each other. Thus, it is not necessary for each function to be available at all times. Instead, the control section of the program, called the root, loads each function only when needed. The root remains in memory at all times.

When a program is segmented, functions may exist within different segments. When the program is executed, only the root segment is initially loaded into memory. Then, when the root calls a function that is not located within a segment that has already been loaded, the segment that contains the function is automatically loaded into memory from disk. The segment remains in memory until the root segment explicitly removes it.

In reality, any reference from one segment to another segment causes the specified segment to be loaded, if it was not previously loaded in. Therefore, no restriction is imposed upon the structure of the program. In fact, one way to understand segmentation is to think of it simply as delayed program loading. If no segment is ever freed, it is possible to load all segments into memory at once if sufficient memory is present.

Loading a Segment

When a segment is loaded, the call to the segment loader tells it what segment is desired. The segment number is used to index into the segment

data table and get the file offset of the segment in the output file. The segment loader seeks to the offset and calls the DOS loader to load the segment. The hunks in the segment are added to the segment list of the program. "Hunk" is a subdivision of a segment.

Once the segment is loaded, the segment loader walks through the segment Jump Table, changing each entry into an absolute reference. It uses the hunk data tables to find the correct load address and the number of entries per hunk. See the *AmigaDos Technical Reference Manual* for more details on hunks and the Amiga executable file format.

Unloading a Segment

When a segment is unloaded, the above process occurs in reverse. The absolute jumps are converted back to the bsr and offset, the hunks of the segment are removed from the segment list, and the segment is unloaded.

Segments are only unloaded by manual calls from the program to unload a segment, using the **freeseg()** call. The argument is the address of a function in the segment.

For example, if **foo()** is declared to be a function in the segment to be unloaded, the call to **freeseg()** looks like:

```
int foo();  
freeseg(foo);
```

Segload

The segment loader itself is a routine that has the name **.segload**. When you specify segmentation, the linker searches for the **.segload** routine. A default version of **.segload** is supplied in the **c.lib** library as well as in the **lib** directory in the file **segload.o**.

Because **.segload** is not directly callable from a C routine, there is also a user callable routine, **segload()**, which takes as its argument the name of a function within the required segment. **segload()** then loads the appropriate segment into memory without actually calling a function in the segment. This can be used by a program that wants a number of segments to be loaded at the beginning of the program before any other activity occurs.

Linking and Startup Code

When segmented code is used, remember that the startup code for the program must always be in the root segment. If it is not, an error will occur in the linker. The startup code may be forced into the root segment in two ways—by either linking the `c.lib` library into the root segment, or linking just the individual module which contains the startup code into the root segment.

To place the user-referenced library functions into the root segment after all the other segments are linked, use

```
ln prog.o +o@ -lc
```

where `prog.o` represents your own object modules.

To put just the startup code into the root segment and not the entire library, link in either the `crt0.o` or `lcrt0.o` files (for either small code, small data or large code, large data, respectively) like this:

```
ln +o prog.o +o@ crt0.o +o -lc
```

(More about linking may be found under "Code Segmentation".)

Segment Construction

All references between segments are made through a segment Jump Table located in the data hunk. Initially, the reference contains the following 8 bytes:

```
bsr      .segload    ;segment loader routine  
dc.b     segnum      ;number of segment to load  
dc.b     hioffset    ;high 8 bits of 24 bit offset  
dc.w     looffset    ;low 16 bits of 24 bit offset
```

After the segment is loaded, each entry looks like:

```
jmp      absolute    ;relocated address of symbol  
dc.w     offset      ;saved offset to .segload
```

Segment Data Table

The segment data table has two parts. The first part consists of eight bytes each of segment data:

```
dc.l      seg_offset    ;offset to segment in file  
dc.w      tabl_offset   ;offset of segment's entries  
dc.w      hunk_offset   ;relative offset of hunk data
```

Hunk Data

Following the segment data is the hunk data which consists of:

```
dc.w      hunk_num     ;number of the hunk  
dc.w      num_symbol   ;number of symbols in Jmptab
```

The hunk data for each segment is terminated by a long word of zero.

SEGMENTED CODE

As stated, with Aztec C, you can create and execute programs that are larger than available memory by dividing a program's executable code into several segments. A segment is brought into memory when needed; when it is no longer needed, the memory it occupied can be released and reused.

This section describes the creation and use of segmented programs. It is divided into the following paragraphs:

- Programmer Information. Describes how you write programs having segmented code.
- Operator Information. Describes how you link programs having segmented code.

Programmer Information

There are two areas of concern for programs whose code is segmented: how segments are loaded and unloaded, and how programs access global and static data.

SEGMENTATION already described how to load and unload programs. Here we will deal with **Global and Static Data**.

Global and Static Data

There is only one global and static data segment for a program, regardless of the number of code segments it has. This segment is loaded automatically along with the program's root segment, and it remains in memory during the entire execution of the program, independent of the state of the program's code segments.

The name of each global variable in a program is unique: If several segments declare a global variable having the same name, they will both access the same variable.

Operator Information

All of a program's code segments are created during a single activation of the Linker.

The code for a command program can be divided into a maximum of 256 segments, each of which has an identifying number between 0 and 255. All command programs must have a code segment 0, which is the first segment loaded for the program.

The linker command that is used to create a command program whose code is segmented is identical to that which is used to create a command program whose code is unsegmented, except that the list of files are interspersed with the +o option. The +o option causes the object modules that follow it to be placed in a selected code segment.

A segment number can optionally be appended to option +o, to explicitly select the segment into which the following modules are placed. If a segment number is not specified for option +o, the modules are placed in the next available segment.

Example

The following command creates the command program **prog**, which has three code segments. Segment 0 contains the code for the modules **menu.o**, **subs.o**, and any needed modules from **c.lib**. Segment 1 contains the code for the modules **mod1.o** and **mod2.o**. Segment 2 contains the code

for **mod3.o** and **mod4.o**, and any **c.lib** modules referenced by segments 1 and 2 which are not in segment \emptyset :

```
ln -f prog.lnk
```

where **prog.lnk** contains:

```
+to $\emptyset$  menu.o subs.o -lc  
+to mod1.o mod2.o  
+to mod3.o mod4.o -lc
```

All the files for this example could have been specified on the command line that activated the linker. We did not do this for two reasons: First, the entire command would not have fit on one line of this page. Second, you will also want to use **-f** files to link programs that have segmented code, since such programs usually have many modules, making it impractical to specify all the file and segmentation information on one line.

Including Modules from Libraries

This example illustrates a point about library searches during the linking of command programs having segmented code. Library code is always contained in the segment where the library is defined, not where it is referenced. For instance, if a module **x.o** in segment 2 references a function in **c.lib**, which is linked as **+to \emptyset -lc**, than the referenced function will be placed in segment \emptyset .

However, in the above example we do not wish to have an overly large segment \emptyset , so the **c.lib** library is linked in twice. Therefore, when the **menu.o** and **subs.o** modules are linked, only the routines within the **c.lib** library that they reference will be contained within segment \emptyset . When any of the other object modules reference functions with **c.lib**, those functions will be placed into segment 2.

Reselecting Segments

The Linker allows individual segments to be selected more than once on the link line. In this case, the modules specified following the reselection are appended to the code that is already in the segment. A segment can be reselected any number of times.

For example, the above example can be modified so that all c.lib modules referenced by all segments are included in segment \emptyset , by modifying prog.link as follows:

```
+o menu.o subs.o
+o mod1.o mod2.o
+o mod3.o mod4.o
+o $\emptyset$  -lc
```

The +o \emptyset reselects segment \emptyset , so that the modules pulled from c.lib are included in segment \emptyset .

Object Module Libraries

Several libraries of object modules are provided with Aztec C. The base libraries are:

- c.lib nonfloating point functions
- mf.lib Motorola Fast Floating Point (32-bit) math functions
- ma.lib Amiga IEEE floating point (64-bit)
- m.lib Manx IEEE floating point (96-bit)
- m8.lib 68881 math library
- s.lib screen functions

There are several versions of each library, depending on code, data, and integer size. Libraries which use 32 bit ints have the number 32 in their name; libraries which use large code/large data have the letter "l" in their name; libraries which make use of the 68881 coprocessor have an 8 in their name. For example, the large code/large data c.lib library is called cl.lib, and the large code/large data, 32 bit int c.lib library is called cl32.lib.

Assembly Language 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 assembly language statements within C language functions.

CONVENTIONS FOR C CALLABLE, ASSEMBLY LANGUAGE FUNCTIONS

A C callable, assembly language function must obey the conventions that are described in the following paragraphs.

Names of External Functions and Variables

The C compiler translates the name of a function or variable to assembly language by truncating the name to 31 characters and prepending an underscore character. Thus, assembly language modules that are to be accessible from C language modules or that are to access C modules must obey this convention.

For example, the following C language module calls the function `bmp()`, which simply adds 10 to the global short count. A C language module refers to this function as `bmp()`, and an assembly language module refers to it as `_bmp`.

```
int count;
main()
{
    bmp();
}
```

An assembly language version of `_bmp` could be:

```
#asm
    dseg
    public _count
    cseg
    public _bmp
_bmp:
    add.w #10,_count
    rts
    end
#endasm
```

Note: **#asm** and **#endasm** are needed in order to incorporate assembly language routines into C code. See the section **Embedded Assembler Source** in this chapter for more information.

Function Calls and Returns

The assembly language code generated by the compiler for a C language 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 assembly language function returns to a C function caller by issuing an **rts** instruction and leaving the caller's arguments on the stack. The caller then removes the arguments from the stack.

A function returns an integer, pointer, or Fast Floating Point value in register **d0**. IEEE Floating Point values are returned in the register pair **d0/d1**, unless using the 68881 which uses **fpo**.

For example, consider the following assembly language 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 language function refers to this function using the name **sub**.

```
cseg
public      _sub
_sub:
    move.w      4(sp),d0      ;get first argument
    sub.w       6(sp),d0      ;subtract 2nd 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);
}
```

Global Variables

A C module's global variables are in either the uninitialized data segment or in the initialized data segment.

An assembly language 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[];
```

An assembly language module can create an initialized variable that can be accessed by a C function using the public directive and the dc directive. 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.1       str
```

To access `_ptr`, a C function could use the following declaration:

```
extern char *ptr;
```

An assembly language module can access global initialized or uninitialized variables that are created in C modules by defining the variables using a public directive that is in 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 assembly language module can access these variables by using the following directives:

```
dseg  
public      _count, _total
```

The above discussion assumes that the C modules and the assembly language modules follow the standard rule in the C language regarding external variables. This rule requires a global variable to be defined without the **extern** keyword in exactly one module and with that keyword in all other modules. Aztec C also supports a relaxed version of this rule. For information on this, see the **Compiler** chapter.

Register Usage

Register usage is implemented according to the Amiga guidelines. An assembly language function that is called by a C function must save and restore all registers it uses, except for **d0**, **d1**, **a0**, and **a1**. These are compiler temporary registers which may be used without being saved.

Embedded Assembler Source

Assembly language statements can be embedded in a C program between an **#asm** and an **#endasm** statement. The pound sign (#) must stand in column one of the line, and the letters must be lowercase.

Embedded assembler code must preserve the contents of all registers it uses, except for **d0**, **d1**, **a0**, and **a1**.

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. Global variables may be referenced as described above.

Support is also provided to access predefined macros, function arguments, local variables, and file scope static identifiers. To access these from within a **#asm** block, precede the identifier name with **% %**. When passing output to the assembler, the compiler checks for the pattern **% %ident**.

If found, it checks to see if *ident* matches any predefined macros and replaces them with the actual constant values. It next checks for matches with any function arguments, local variables, or file scope static variables and replaces them with the appropriate address. If the identifier is not a macro, argument, auto, or static, the two percent characters and identifier are passed to the assembler output file.

In general, it is safest to contain assembly code in a separate, assembly language module rather than embedding it in C source.

Interrupt Service Routines

It is possible to write interrupt service routines (interrupt handlers) using Aztec C, without resorting to assembly language. Interrupt routines provide real-time response to events. Interrupts are serviced by the Amiga Exec system routines and, if an application program specifies, an interrupt service routine can be called in response to a particular interrupt.

When the interrupt occurs, the processor can be just about anywhere, with almost anything in its registers. The system interrupt handler preserves a number of registers and then calls the user service routine. However, it does not save all of the registers. In particular, it does not save registers **d2**, **d3**, and **a4**.

If the interrupt service routine modifies these registers, it must first preserve them. Since Aztec C considers **d2** and **d3** to be temporary registers, it does not preserve them. In addition, register **a4** is used as a base register to the global data of the program and should be modified if small code and data are being used.

There are two simple ways of preserving the necessary registers. First, if the program is compiled with option **-mcd**, the compiler preserves **d2** and **d3** at the entry to every function. In addition, this option forces the large code and data model. Therefore, register **a4** need not be modified and need not be saved. Note that the object modules need to be linked with the **c1** library.

The second way is necessary if you are not compiling with the **-mcd** option. Essentially, you should call two routines in the library at the beginning and end of your interrupt code which will save the necessary registers as well as setting up the **a4** register for small data references. The functions are called **—int_start()** and **int_end()**.

As an example, the interrupt handler **intfunc** looks like this:

```
intfunc()
{
    int_start();
    ...
    int_end();
}
```

The reason the **a4** register may sometimes need to be saved involves the different memory reference models allowed by Aztec C. In the large model, all references are absolute and 32 bits. In the small model, references are 16 bits and either pc-relative or relative to some base register. For Aztec C, the base register is **a4**.

Therefore, all references are made relative to register **a4**. When the interrupt occurs, register **a4** might be pointing anywhere because almost any task might be executing, not only ours. If **a4** is not set up correctly by the interrupt service routine as the very first thing, the interrupt routine might use the **a4** register to access some global variables, which could create all sorts of problems.

Creating SubTasks

The Amiga's multi-tasking capability is very useful, since it is often easier for a separate task to handle asynchronous events without tying up the

main program. An excellent example of this is the **beep** program in the examples directory. A secondary task is created to actually make the beep sound whenever it gets a message from the parent task.

Creating the subtasks is discussed in the *Amiga Rom Kernel Manual* to some extent. However, there are a few special considerations when using Aztec C. In particular, when using small code and data, there is a little extra setup that must be done. In this case, the **a4** register is used as a base register to access global variables. Thus, for the program to operate correctly, **a4** must point at the proper memory location.

When a task is started, the registers contain somewhat arbitrary values. It is almost certain that **A4** will not contain the appropriate value. Therefore, the very first thing the task should do is set up the **a4** register. This is done by calling the routine **geta4()** as the first thing in the new task's function. For example:

```
main()
{
    void subtask();
    CreateTask("TASK", (long)PRI, subtask,
               (long)STKSIZ);
    ...
}

void subtask()
{
    geta4();
    ...
}
```

Floating Point Information

There are three floating point formats supported by Aztec C68k on the Amiga. Which one you choose depends on whether you are looking for speed, accuracy, or whether you have a math coprocessor on your machine.

HOW TO CREATE A FLOATING POINT FORMAT PROGRAM

When using floating point, there are two rules that **must** be followed for your program to work correctly.

- Compile with the desired floating point option
- Link with the corresponding floating point library and place that before the c.lib

FLOATING POINT FORMATS

The compiler generates code to use the appropriate format based on one of the following options: **-ff**, which is the Motorola Fast Floating Point; **-fa**, which is the Amiga IEEE Double Precision Floating Point Emulation; **-fm**, which is the Manx IEEE Double Precision Floating Point Emulation; and **-f8**, which is the 68881 Floating Point. These are all described in detail in the **Compiler** chapter.

The MathFFF library and the MathIeeeDoubBas library are opened on the first access of a floating point function when using the **mf.lib** or **ma.lib** library.

For example, if a program wished to use the Amiga IEEE format to achieve better floating point precision, the program would be compiled and linked this way:

```
cc -fa prog.c  
ln prog.o -lma -lc
```

Building Programs that Run from Workbench

OVERVIEW

This section discusses how to build programs that run from the Workbench.

First, however, read the *Amiga ROM Kernel Manual* which discusses the Workbench.

There are two ways for a program running from the Workbench to operate. First, the program can handle all its own I/O by creating windows, attaching a console device to the window, and getting input from the console and/or Intuition. In this case, there is not much to discuss because everything is handled by the programmer.

The second way is for the program to use the standard input and output channels that are assumed by the program to be preopened by the startup code. In this case, the program is doing most or all of its operations as though it were running on a simple line-oriented terminal. The trick here is to get a window opened by the startup code. This is accomplished by the `_wb_parse()` routine, which scans the Tool Array types, which are part of the Icon used to invoke the tool. It scans the array for a definition like the following:

WINDOW=xxxx

If found, the routine attempts to open the specified string and, if successful, sets `pr_CIS` and `pr_COS` to the window. Then, `_main()`, `stdin`, `stdout`, and `stderr` are set up for the window as well. If not found, `stdin`, `stdout`, and `stderr` are not initialized and any output is thrown away.

Note that this allows for debugging messages to be placed in a program and only displayed when the appropriate ToolType is set.

The only other note of importance is that when a program runs from the Workbench, the initial Workbench startup message is passed as `argv` to `main` with `argc` set to zero. However, for compatibility, the global variable `WBenchMsg` is initialized to point to the startup message as well.

STEP BY STEP

This section describes the step by step procedure to produce a program that displays the "Hello world!" message from the Workbench. Boot the

system using the **sys:1** disk and place the **sys:2** disk in the second drive.
Type the following commands:

```
cd ram:  
copy * hello.c  
  
main()  
{  
    printf("Hello world!\n");  
    getchar();  
}  
^\  
cc hello.c  
ln hello.o -lc
```

At this point, run the **hello** program just to make sure it works from the CLI. The reason the **getchar()** routine is called is to give you time to see the message when you run from the Workbench.

Type:

iconed

The **IconEd** program starts and displays a screen with some information. Click the **OK** button and, in the menu, select **clear frame** to erase the existing picture. You may exercise your artistic freedom and make any kind of icon.

Once you have your icon designed, select **save** from the menus. A small requester appears. Change the filename to "hello". Click in the "Save entire image" box and close the window to exit the program. Type the following line:

loadwb

This command starts up the Workbench, so resize the CLI's window until you can see the disk icons. Click on the **Ram:** icon and you should see your

Hello icon. Click on this icon *once*. This selects it but does not attempt to execute it. Now, from the WORKBENCH menu, select INFO.

Selecting INFO should bring a window up with a number of gadgets and some information about the icon. Toward the bottom of the window is a single line: "TOOL TYPES string gadget". Click on the ADD button, click in the string gadget and type the following line:

```
WINDOW=CON:0/0/500/30>Hello
```

Then click on the ADD button again, and then click on the SAVE button in the bottom left-hand corner of the window.

That is all there is to it! Double click on the Hello icon now and the window should appear with the message waiting for a line to be typed. Press <CR> to end the program and the window disappears.

ERROR MESSAGES

9

Chapter 9 - Error Messages

This chapter discusses error messages that can be generated by the compiler, assembler, and linker. It is divided into three sections: the first summarizes the compiler messages, the second the assembler, and the third discusses linker error messages.

A complete list of all error messages is provided on the next few pages. The error message explanations begin on page 9-11.

List of Error Messages

Compiler Error Messages

```
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: item not previously declared found in prototype
```

Fatal Compiler Error Messages

In the following "%%" is used to indicate a variable number of items can appear at that place.

```
out of memory!
unable to execute %%
too many -i options
illegal '-%%' option: %%
illegal '-h' option: '%%'
multiple '-h' options specified
more than one output file name specified
illegal option: '%%'
illegal 3.6 option: '%%'
illegal 3.6 '+x' option: '%%'
multiple input files specified!
can't open file '%%' for input!
no input file was specified!
cannot create output file '%%'!
too few arguments for '%%' option!
dump file not found!
pre-compiled header not in proper format!
pre-compiled header uses wrong size int!
error reading dump file!
error creating dump!!
Internal Errors
```

```
attempt to release item already free
attempt to use expression tree entry twice
optim failure
out of registers!
bad arginfo
bad op in cgen: 'op'
bad cast - 'cast'
```

Assembler Error Messages

Opcode Error Messages

```
addressing mode not valid
```

Need data or address register as index
Only W and L are valid modifiers
Scale must be 1, 2, 4, or 8
Field width must be in range (0-31)
Need data register here
Missing ':' in bit field syntax
Field width must be in range (1-32)
Missing ')' in bit field syntax
Need opcode, directive or macro name here
Unimplemented opcode
Size extension not allowed on this opcode
Post incrementing illegal in this mode
second argument must be immediate
Need register list as one of the operands
Need FP register list as one of the operands (mc68881)
An, Dn, modes not supported
illegal function code (mc68851)
pmove PSR, (ea) not allowed (mc68851)
Opcode extension size did not match
Bad size for expression
Opcode operands did not match
Illegal addressing format
Illegal argument expression
must be 16 bit
must be 8 bit
Need data register here
Missing argument
Bad argument
Invalid argument

Directive Error Messages

Illegal character in directive arguments
Equate directive requires a label
Invalid expression for equate directive

Need absolute value here
Set directive requires a label
Invalid expression for set directive
Name already defined
Multiply defined symbol
Register list directive requires a label
Need register list here
REG directive must contain register list
Macro end out of place
Need symbol for definition check
Requires two strings
Right side of test must match left side
Illegal character in conditional
Premature end of line in conditional string
Conditional end directive out of place
Need file name here
Unable to open include file
Need 'code' or 'data' here
Need second argument as well
Invalid expression for defined storage
Unimplemented assembler directive

Syntax Errors

bad '('
Missing ')'; Where's the ')'; Missing trailing
parenthesis;
Missing ']'; illegal ']';
illegal '['
bad syntax
extra characters on line!
Unknown token in expression
Unknown opcode or directive
Illegal character in string
Bad character in line

Linker Error Messages

Command Line Errors:

```
Unknown option <bad option letter>
Too few arguments in command line
No input given!
Cannot have nested -F options.
Too few arguments in -F file: <filename>
Multiple entry points defined.
Invalid overlay number
```

I/O Errors

```
Cannot open <filename>, err = <errno>
Cannot open -F file: <filename>
I/O error (<errno>) reading/writing output file
Cannot write output file
Cannot create output file: <filename>
Cannot create symbol table output
Error creating .map file
Cannot create debugger symbol file<filename>
Error creating symbol listing file
Corrupted object files
Object file is bad!
Invalid operator in evaluate <hex value>
Library format is invalid!
Cannot read module from <input> on pass2
    or cannot find symbol, symbol name, on pass 2
Not an object file
Bad symbol typing information
Line displacement too large
Symbol name too long
```

Errors in Use of Memory

Insufficient memory!
Too many symbols!

Errors Arising From Source Code

```
Undefined symbol: <symbol name>
<symbol name> multiply defined
pass1(<hex value>) and pass2(<hex value>) values
    differ:
or symbol type differs on pass two: <symbol name>
Branch out of range @pc=<addr>
Short branch to next location @pc=<addr>
Entry point must be in root segment
Entry point must be in first 64K of program
Attempt to store out of bounds
Program is too large to link
Attempt to perform relocation in overlay code
data ref to overlay code not in jump table
BigCrt0.o must be in root segment
Absolute reference from segment to segment
Excess number of segments
Can't do relocation from data to nonroot code
Data +BSS +JumpTab = 64K (code resource)
data reference out of range—remake with large data
    model
```

Compiler Error Messages

Note:

- Error codes 116 and higher will not occur on Aztec C compilers whose version number is less than 3.
- 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, 0xFFFF).

2: **string space exhausted**

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option -z. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program.

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; /* misspelled 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 declara-

tion 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.

<i>data type</i>	<i>interpretation</i>	<i>size(bytes)</i>
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

Automatic arrays and structures may not be initialized. Static and external aggregates may be initialized, but by default their members are set to zero.

```
char array[5] = { 'a', 'b', 'c', 'd' };
function()
{
    static struct suzanne {
        int bone;
        char license[10];
    } daniel = {
        1,
        "123-4-1984"
    };

    char autoarray[2] = { 'f', 'g' }; /* no good */
    extern char array[];
}
```

There are three variables in the above example, two of which are correctly initialized. The variable **array** may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.

The structure **daniel** is static and may be initialized. Notice that **license** cannot be initialized without first giving a value to **bone**. There are no provisions in C for setting a value in the middle of an aggregate.

The variable **autoarray** is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic aggregates cannot be initialized.

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] = "aBDdefghij";
```

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 the Amiga 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 occurrences 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, conditional 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 accidentally 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,*)* error 98 */  
func(exchange(i));
```

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 statements 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 augmented 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 */
#endif
```

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: 1. following a question mark as part of a conditional, as in `(flag ? 1 : 0)`; 2. 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;
*c=0; /* warning 125, because 0xc 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 the Amiga 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 the Amiga 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(AMIGA) /* 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 the Amiga 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 the Amiga 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 the Amiga 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 func-

tions, 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 t he 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: item not previously declared found in prototype

This warning message will be generated if a `struct` appears as an argument in a function prototype and there is no previous declaration for the `struct`. This warning will be generated if there is no `struct` declaration or if the `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 {
    int a;
    char c;
}

void func(struct astruct arg);

void func(struct astruct arg);
{
...
}
```

This problem presents special difficulties when it arises because the intended `struct` decalartation occurs after the prototype definition. A strict interpretation of ANSI rules, in this case, produces results that may seem arbitrary and illogical. Positioning the `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.

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.

out of disk space!

There is no room on the disk for the output file of the compiler. Previous disk files will not be overwritten by the compiler's assembly language output. To make room on the disk, it is usually sufficient to remove unneeded files from the disk.

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.

multiple '-h' options specified

Only one `-hi` or `-ho` option can be 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.

duplicate output file

If an output file name has been specified with the **-o** option and that file already exists on the disk, the compiler will not overwrite it. **-o** must specify a new file.

too few arguments for -o option

The compiler expected to find the output filename 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.

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.

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.

multiple input files specified

More than one input file was specified

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.

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.

no 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.

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.

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.

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.

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.

duplicate symbol dump file

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.

unable to execute %%

The compiler can chain to other programs such as the assembler or editor. There are various environment variables and option specification that determine the programs the compiler should launch and where it should look for them. Either the program does not exist on the disk, or the compiler was misled as to its whereabouts.

too few arguments for -%% option

The option expects a filename to follow.

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.

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.

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.

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.

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.

Internal Errors

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
optim failure.
out of registers!
bad op in cgen: 'op'
bad cast - 'cast'
```

Assembler Error Messages

This section describes the error messages that the Assembler generates as it is creating an object module.

Opcode Error Messages**68020 addressing mode not valid**

To enable assembly of 68020 addressing modes insert the directive:

```
machine mc68020
```

**Need data or address register as index
Only W and L are valid modifiers
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.)

**Field width must be in range (0-31)
Need data register here
Missing ':' in bit field syntax
Field width must be in range (1-32)
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).

Need opcode, directive or macro name here

Label names must be defined starting in the first column of the line.

Unimplemented opcode

Assembler does not understand opcode. Check your syntax.

(Note: The mnemonic TRAPcc has been changed to Tcc in the parameterless form, and TPcc is used when immediate data is specified. Similar changes have been made for FTRAPcc (mc68881) and PTRAPcc (mc68851).)

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

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

second argument must be immediate

Should only occur when using pflush (mc68851). The mask must be immediate.

Need register list as one of the operands

A register list is used with movem.

Examples:

d3-d7	means d3, d4, d5, d6, and d7
d5/a5	means d5 and a5
d2-d4/a0-a3	means d2, d3, d4, a0, a1, a2, and a3

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
```

An, Dn, modes not supported

Certain 68851 opcodes will not accept **an** or **dn**. (See your 68851 reference manual).

illegal function code (mc68851)

Function codes must be expressed in any of the following registers:
dn, sfc, dfc, or as an immediate four bit number.

Examples:

```
ploadr d4, (a5)  
ploadr sfc, (a5)
```

pmove PSR, (ea) not allowed (mc68851)

(consult your 68851 reference manual)

**Opcode extension size did not match
Bad size for expression**

Occurs when you try to use an illegal size extension.

**Opcode operands did not match
Illegal addressing format
Illegal argument expression**

The address mode you are trying to use is not supported by the opcode.

must be 16 bit

Data must be 16 bit (\$0000 - \$ffff)

must be 8 bit

Data must be 8 bit (\$00 - \$ff)

Need data register here

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.)

Missing argument

Directive argument is expected.

Bad argument**Invalid argument**

Legal directive argument is expected.

Directive Error Messages

Illegal character in directive arguments

Directives must be carefully described. (See the Assembler chapter.)

Equate directive requires a label

A label must precede all equate directives.

Example:

```
$4000 equ tempvar ;    ERROR
tempvar equ $4000 ;    OK
```

**Invalid expression for equate directive
Need absolute value here**

An absolute value is a constant expression.

An absolute value must follow a set directive.

Examples:

<code>rex equ 1 ;</code>	OK
<code>spot equ 2 ;</code>	OK
<code>fido equ rex+spot ;</code>	OK
<code>rover equ rex+(a4)</code>	ERROR

Set directive requires a label

(see: Equate directive requires a label)

Invalid expression for set directive

(see: Invalid expression for equate directive)

**Name already defined
Multiply defined symbol**

Every label must have a unique name.

**Register list directive requires a label
Need register list here
REG directive must contain register list**

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.

Macro end out of place

The ENDM directive must be used in conjunction with the MACRO directive. All directives and opcodes must be preceded by at least one *space* or *tab*.

Need symbol for definition check

Requires two strings

Right side of test must match left side

Illegal character in conditional

Premature end of line in conditional string

These errors pertain to conditional directives.

Conditional end directive out of place

endc must be used in conjunction with an if directive.

Need file name here

Unable to open include file

Include directives must be specified as:

include <filename>
include "filename"

Need 'code' or 'data' here

The **near** and **far** directives must specify 'code' or 'data'

Examples:

near code
far data

Need second argument as well

The **cnop** directive requires two arguments.

Invalid expression for defined storage

The value of the size field in the ds directive must be an absolute expression.

Unimplemented assembler directive

The assembler does not understand. Check your syntax.

Syntax Errors

```
bad '('
Missing ')'; Where's the ')'; Missing trailing parenthesis;
Missing ']'; illegal ']';
illegal '['
bad syntax
extra characters on line!
Unknown token in expression
Unknown opcode or directive
Illegal character in string
Bad character in line
```

Please check your typing.

Linker Error Messages

This section discusses the error messages that the linker may display as it creates an executable program. Below is a summary of the error messages, followed by a more detailed explanation of each.

Note: Only one file type option is permitted.

Explanation of Linker Error Messages

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. In generates an executable program in two stages, known as pass 1 and pass 2. The size messages are printed at the end of pass 1, so any errors occurring after that are detected during pass 2 of the linker.

Following is a list of the messages that the linker generates in response to an error. The messages are grouped according to the source of the errors that cause them. Elements that are variable are enclosed by angled brackets.

Command Line Errors

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.

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.

No input given!

The linker quits immediately if it is not given any input to process.

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

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.

Multiple entry points defined.

Multiple global symbols have been found that have the same name.

Invalid overlay number

Overlays must be positive integers.

I/O Errors

Cannot 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*.

Cannot open -f file: *filename*

A file given with option -f cannot be opened.

I/O 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.

Cannot write output file
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
Error creating .map file
Cannot create debugger symbol file *filename*
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.

Corrupted object files
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.

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.

**Cannot read module from input on pass2
...or cannot find symbol, symbol name, on pass two**

Either message indicates that a module is corrupted between pass 1 and pass 2. 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 a hardware failure.

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.

Bad symbol typing information

Line displacement too large

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

Insufficient 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

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.

***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.

**pass1(*hex_value*) and pass2(*hex_value*) values differ:
or 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.

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.

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.

Entry point must be in root segment

The entry point for a program must be in the program root segment. For example, this error would be generated if you tried to link the "hello, world" program with the command:

```
ln hello.o +0 -lc
```

Entry point must be in first 64K of program

Not only must a program's entry point be in the root segment, it must also be in the first 64k bytes of this segment. The reason for this is that a 16-bit field in the jump table points to the entry point.

Attempt to store out of bounds

This error should not occur. It indicates a linker bug.

Program is too large to link

This error should not occur. It indicates a linker bug.

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.

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.

BigCrt0.o must be in root segment

Attempt to link large data or large segments without appropriate starting code. Place **BigCrt0.o** early in your **ln** command.

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.

Excess number of segments

Linker internal overflow. Try using fewer and larger segments.

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.

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.

Index

#asm/ #endasm
 See embedded assembly
#include files
 pre-compiled, 2-6 - 2-7
 search order, 2-6
 searching for, 2-5
#pragmas
 See pragmas
* operand as argument to print, db, 7-35
.dbg file, 5-16
.dbinit file, db, 7-7
== command, db, 7-43
? command, db, 7-44
@function, db, 7-12
 __FILE__, 3-13
 __FUNC__, 3-13
 __LINE__, 3-13
 __INT32, 3-14
 __LARGE_CODE, 2-26, 3-14
 __LARGE_DATA, 2-26, 3-14

A

abnormal termination of program, db, 7-4
accessing variables
 from C and assembly, 8-14
add command, db, 7-14
ADDR
 definition, 7-11
 examples, 7-13
addressing, db commands, 7-15
adi command, db, 7-14
adl command, db, 7-14
adp command, db, 7-14
adr command, db, 7-14
adump utility, 6-3

aggregate types
 arrays, 3-10
 structures, 3-10
ai command, db, 7-15, 7-32
ak command, db, 7-15
al command, db, 7-16
am command, db, 7-17
amicall
 See pragmas
Amiga ROM Kernel Manual, 8-18
Amiga specific commands, db, 7-14
an command, db, 7-17
ANSI language specifications, 3-1
 examples, 3-8
 auto aggregate initialization, 3-10
 extended syntax, 3-7
 features supported, 3-6
 function prototypes, 3-6
 K&R versus ANSI, 3-7
 overview, 1-1, 2-36
 predefined symbols, 3-13
 See also prototypes
 size_t, 3-13
 sizeof and size_t, 3-13
 trigraphs, 3-11
 wide strings and literals, 3-10
ANSI preprocessor features
 line continuation, 3-15
 predefined symbols, 3-13 - 3-14
 string concatenation, 3-15
ap command, db, 7-17
aq command, db, 7-18
ar command, db, 7-19
arcv utility, 6-2
array, initialization, 3-10
as command, db, 7-19
assembler
 definition, 4-1
 syntax, 4-2
 accessing modules, 8-12

arguments to subroutines, 4-21
embedded with C, 2-44, 4-21 - 4-22
See also embedded assembly
executable instructions, 4-8
execution environment, 4-2
file creation, 2-4
functions callable from C, 8-12 - 8-16
INCLUDE environment variable, 4-4
include files, 4-4 - 4-5
input file, 4-2
listing file, 4-3
memory models, 4-5
object code file, 4-3
operating instructions, 4-2
optimizations, 4-3 - 4-4
output control, 2-15
processor support, 4-2
register conventions, 4-21
searching for include files, 4-4
source program structure, 4-8 - 4-14, 4-16, 4-18 - 4-20
source statements, 4-1
using with C source code, 2-44 - 2-45, 8-12, 8-15
assembler directives, 4-10, 4-12 - 4-14, 4-16, 4-18 - 4-20
 blanks, 4-10
 bss, 4-16
 clist, 4-10
 cnop, 4-11
 cseg, 4-11
 dc - define constant, 4-12
 dcb - define constant block, 4-12
 ds - define storage, 4-13
 dseg, 4-11
 else, 4-15
 end, 4-13
 endc, 4-15
 endm, 4-17 - 4-18
 entry, 4-13
 equ, 4-13
 equr, 4-14
 even, 4-14

fail, 4-14
far code, 4-14
far data, 4-14
freq, 4-14
global, 4-16
if, 4-15
ifc, 4-15
ifd, 4-15
ifnc, 4-15
ifnd, 4-15
include, 4-16
list, 4-17
machine, 4-17
macro, 4-17 - 4-18
mc68851, 4-18
mc68881, 4-18
mexit, 4-18
mlist, 4-17
near code, 4-16
near data, 4-16
noclist, 4-10
nolist, 4-17
nomlist, 4-17
other ifs, 4-16
public, 4-19
reg, 4-19
section, 4-19
set, 4-19 - 4-20
ttl, 4-20
xdef, 4-20
xref, 4-20

assembler error messages

opcode summary list, 9-7 - 9-9
explanations, directive errors, 9-66 - 9-69
explanations, opcode errors, 9-62 - 9-66
explanations, syntax errors, 9-69

assembler options

summary list, 4-6
-c, 4-5, 4-7
-d, 4-5, 4-7

- i, 4-4 - 4-5, 4-7
- l, 4-3, 4-7
- n, 4-3
- o, 4-7
 - o example, 4-3
- at command, db, 7-19
- auto aggregate initialization, 3-10
- available memory
- segmentation, 8-8

B

- backtracing, db, 7-6
- bb command, db, 7-20
- bc command, db, 7-21
- bd command, db, 7-21
- bh command, db, 7-22
- bitfields, 3-5
- bl command, db, 7-20
- bq command, db, 7-22
- br command, db, 7-23
- branches
 - optimizations, 4-3
- breakpoints, db, 7-4
 - commands, 7-5
 - skip count, 7-4
- bs command, db, 7-23
- bt command, db, 7-24
- bu command, db, 7-24
- bw command, db, 7-20

C

- C and assembly
 - accessing variables, 8-14
 - stack operations, 8-13
- c.lib, 8-11
- CCOPTS environment variable
 - See environment variables
- char type, 3-4
- characters

- long constant, 3-12
- non-alphanumeric, 3-11
- wide, 3-10
- chip or processor control, 2-15
- clear commands, db, 7-25
- cmdlist, definition, 7-14
- cmp utility, 6-4
- cnm utility, 6-5 - 6-9
 - options, 6-6
 - symbol format, 6-7
 - symbol types, 6-7
 - type codes, 6-7 - 6-9
- code segmentation, 2-14
- code segments, 8-2
 - location in memory, 8-3
 - size, 8-4
- command breakpoints, db, 7-5
- command line errors
 - See linker error messages
- command summary, db
 - See db commands
- compatibility, 3-1
 - Amiga Workbench, 3-1
 - UNIX, 3-1
 - with other Aztec compilers, 3-1
 - XENIX, 3-1
- compiler
 - See also libraries
 - _LARGE_CODE, defining, 2-26
 - _LARGE_DATA, defining, 2-26
 - aborting, 2-2
 - code segment limits, 2-14
 - converting data, 2-43
 - declared functions, 2-41 - 2-42
 - displaying error messages, 2-34
 - embedded assembly, 2-44
 - example, 2-44
 - enums, 2-27
 - error handling, 2-34
 - fatal errors, 2-34

for loops, 2-29
handling errors, 2-34
input files, 2-2
internal storage of numeric data, 2-43
invoking, 2-1
line continuation, 2-44
memory models, 2-8 - 2-14
mixing memory models, 2-13
See also memory models
nonfatal errors, 2-34
output files, assembly language, 2-4
output files, object code, 2-3
pointer considerations, 2-41
pointer use, rules, 2-41
See also pointers
pre-compiled #include files, 2-6 - 2-7
predefined symbols, 3-14
prepended underscore use in db, 7-3
prototypes, 2-28
searching for #include files, 2-5
selecting a memory model, 2-12
syntax, 2-1
table manipulation options, description, 2-34
trigraph option, 3-12
trigraphs, 2-28
utility options - description, 2-23
compiler error messages
 summary list, 9-2 - 9-6
 fatal error summary list, 9-7
 explanations, 9-12 - 9-57
 explanations, fatal errors, 9-58 - 9-62
 explanations, internal errors, 9-62
compiler options, 2-14
 summary list, 2-17 - 2-20
 -3 option, 2-17, 2-21 - 2-22
 -5 option, 2-17, 2-21 - 2-22
 -a option, 2-2, 2-4 - 2-5, 2-17
 -a examples, 2-4
 -at option, 2-4 - 2-5, 2-17
 -bd option, 2-17

- bs option, 2-17, 5-16
- c2 option, 2-17, 2-22
- d option, 2-17, 2-22 - 2-23
- e option, 2-21
- f8 option, 2-17, 2-24
- fa option, 2-17, 2-23
- ff option, 2-17, 2-23
- fm option, 2-17, 2-24
- hi option, 2-7, 2-17, 2-24
 - hi examples, 2-7
- ho option, 2-7, 2-17, 2-24
 - ho examples, 2-7
- i option, 2-5, 2-17, 2-25
- k option, 2-17, 2-21, 2-25
- ma option, 2-17
- mb option, 2-18, 2-26
- mc option, 2-12 - 2-13, 2-18, 2-26
- md option, 2-12 - 2-13, 2-18, 2-26
- me option, 2-18
- mm option, 2-18, 2-26
- ms option, 2-18
- o option, 2-18, 2-27
- pa option, 2-18, 2-27
 - pa examples, 2-15
- pb option, 2-18, 2-27
- pc option, 2-18
- pe option, 2-18, 2-27
- pl option, 2-18, 2-27
- po option, 2-18
- pp option, 2-18
- ps option, 2-18, 2-28, 2-41
 - ps examples, 2-15
- pt option, 2-18, 2-28
- pu option, 2-18 - 2-19
- qa option, 2-19, 2-28
- qf option, 2-1, 2-19, 2-29
- qp option, 2-19, 2-29
- qq option, 2-19, 2-29
- qs option, 2-19, 2-29
- qv option, 2-19, 2-29

- r4 option, 2-19
- sa option, 2-19
- sb option, 2-19
- sf option, 2-19, 2-29
- sm option, 2-19
- sn option, 2-19, 2-30
- so option, 2-19
- sp option, 2-19, 2-30
- sr option, 2-19, 2-30
- ss option, 2-20, 2-30
- su option, 2-20, 2-31
- wa option, 2-20, 2-31
- wd option, 2-20, 2-31
- we option, 2-20
- wl option, 2-20, 2-32
- wn option, 2-20, 2-32
- wo option, 2-20, 2-32
- wp option, 2-20, 2-32
- wq option, 2-20, 2-33
- wr option, 2-20, 2-33
- ws option, 2-20
- wu option, 2-20, 2-33
- ww option, 2-20, 2-34
- z option, 2-21

compiler options, turning off, 2-15

const, 3-5

constant

- definition, 7-10

continuation, line, 3-15

cs command, db, 7-25

current directory

- loading db files from, 7-4

D

- d command, db, 7-26
- data segments
 - location in memory, 8-4
 - size limitations, 8-4
- data types, 3-2
 - bitfields, 3-5

- example, 3-5
- characters, 3-4
 - example, 3-4
- const, 3-5
- enums, 3-5
- integers and long integers, 3-3
 - example, 3-3
- structures, 3-4
 - example, 3-4
- volatile, 3-5

data, converting, 2-43

db

- See also db terms
- .dbinit file, 7-7
- @function, 7-12
- abnormal termination of program, 7-4
- addressing commands, 7-15
- Amiga specific commands, 7-14
- backtracing, 7-6
- basic commands, 7-2
- breakpoint skip count, 7-4
- breakpoints, 7-4 - 7-5
- code symbols, 7-3
- command format, 7-2
- command vs table breakpoints, 7-5
- compiler prepended underscore, 7-3
- creating macros, 7-6
- data symbols, 7-3
- default radix setting, 7-10
- defining breakpoints, 7-4
- desc_code list, print option, 7-37
- display code and data symbols, 7-3
- entering a function, 7-6
- environment variables, 7-7
- examining memory, 7-2
- examining variables, 7-4
- exiting, 7-39
- exiting a function, 7-6
- loading programs, 7-3
- loading programs not in current directory, 7-4

loading symbols, 7-4
macros, 7-6
memory change breakpoints, 7-5
modifying memory, 7-2
overlay breakpoints, 7-8
print command * operand, 7-35
print command format string, 7-33
print count specifier, 7-33
print desc_code, 7-33
program control, 7-4
program termination, 7-4, 7-8
referencing symbols by location, 7-3
referencing symbols by name, 7-3
removing memory breakpoints, 7-6
requirements, 7-1
scatter loading support, 7-8
segment loading and unloading, 7-9
setting memory breakpoints, 7-6
single-stepping, 7-2
skip counter, 7-5
special characters, 7-11
starting db, 7-3, 7-7
startup events, 7-7
stopping display, 7-9
stopping the program, 7-9
symbol table generated by linker, 7-3
symbol table hunks, 7-3
trace mode, 7-6
unassembly, 7-2
viewing code and data symbols, 7-3

db command, db, 7-26
db commands

- summary list, 7-45
 - Special character commands, 7-43
 - < - redirect input, 7-43
 - == - display expression, 7-43
 - > - log output to file, 7-43
 - >> - log commands only, 7-43
 - ? - help, 7-2, 7-7, 7-12, 7-44
- Amiga specific commands, 7-15

add - display device list, 7-14
adi - display interrupt list, 7-14
adl - display library list, 7-14
adp - display port list, 7-14
adr - display resource list, 7-14
ai - display task info, 7-15, 7-32
ak - kill current task, 7-15
al/aL - debug next task w/o symbols, 7-16
am - amt of free memory in system, 7-17
an/aN - new window and symbols, 7-17
ap/aP - debug a crashed program, 7-17
aq - close all windows, 7-18
ar - allow current task to resume, 7-19
as/aS - select a task to debug, 7-19
at - display all tasks, 7-19

Breakpoint commands, 7-2

bb - set byte breakpoint, 7-20
bc/bC - clear breakpoint, 7-21
bd - display breakpoints, 7-21
bh - set memory hash breakpoint, 7-22
bl - set long breakpoint, 7-20
bq - toggle low memory checksum, 7-22
br - reset breakpoint counter, 7-23
bs - set or modify a breakpoint, 7-23
bt/bT - toggle trace mode flag, 7-24
bu - user defined breakpoint, 7-24
bw - set word breakpoint, 7-20

Clear symbol table commands, 7-25

cs - clear symbol table, 7-3, 7-25

Display commands, 7-26

d - display memory in last format, 7-26
db - display bytes in memory, 7-26
dc - display all code symbols, 7-26
dd - display data symbols, 7-27
dg - display global values, 7-27
dl - display memory in longs, 7-26
ds - display stack backtrace, 7-27
dw - display memory in words, 7-26

Memory commands, 7-29

ma - allocate some memory, 7-29

mb - modify bytes of memory, 7-30
mc - compare memory, 7-30
mf - fill memory, 7-31
ml - modify double words of memory, 7-30
mm - move memory, 7-31
ms - search memory, 7-31
mw - modify words of memory, 7-30
 Miscellaneous commands, 7-28 - 7-29
n - change radix, 7-32
g/G - execute the program, 7-28, 7-32, 7-34 - 7-42

db terms

addr, 7-13
addr and *addr, 7-11
cmdlist, 7-14
constant, 7-10
expr, 7-9
period, 7-12
range, 7-13
registers, 7-10
term, 7-9
dc command, db, 7-26
dd command, db, 7-27
debugging control, 2-15
declared functions, 2-41 - 2-42
default radix setting, db, 7-10
default settings, 2-14
defining breakpoints, db, 7-4
desc_code list, db, 7-37
device drivers, 5-1
dg command, db, 7-27
directive error messages
 See assembler error messages
directives, assembler
 See assembler directives
display task information, db, 7-15
dl command, db, 7-26
ds command, db, 7-27
du utility, 6-10
dw command, db, 7-26

E

embedded assembler
 examples, 2-44, 8-12
 source, 8-15
enums, 2-27, 3-5
environment variables
 See also specific variable name
 CCOPTS, 2-14 - 2-16, 2-21 - 2-22
 CCTEMP, 2-4
 CLIB, 5-4, 5-11
 db, 7-7
 displaying, 6-28
 INCLUDE, 2-5 - 2-6, 4-4 - 4-5
 setting, 6-28
error handling, compiler, 2-34
error messages, assembler
 See assembler error messages
error messages, compiler
 See compiler error messages
error messages, linker
 See linker error messages
escape sequences, 3-12
examining variables, db, 7-4
executable file, 5-4, 5-8
executable instructions
 assembler, 4-8 - 4-10
 labels, 4-8
 operands, 4-9 - 4-10
 operations, 4-8 - 4-9
executable programs, 5-1, 5-7
execution environment
 assembler, 4-2
exiting a function, db, 7-6
exiting db, 7-39
EXPR, definition, 7-9
extern keyword
 use with global variables, 8-15
external functions, 8-12

F

- floating point control, 2-15
- floating point support, 8-18
 - compile requirements, 8-19
 - compiler options, 2-23
 - descriptions of, 8-19
 - formats, 8-19
 - link requirements, 8-19
 - register usage, 8-13
- function
 - K&R versus ANSI, 3-7
 - structure arguments, 3-11
- function calls, 8-13
- function calls and returns, 8-13
- function definition, 3-7
- function entering, db, 7-6
- function prototypes, 3-9
 - See prototypes
- functions.h, 2-37 - 2-38

G

- g command, db, 7-28
- geta4 function, 8-18
- global data, 8-9
- global variables, 8-14

H

- hd utility, 6-11
- heap segments, 8-3
 - size of, 8-4
- help command, db, 7-7
- hunk data, 5-14, 8-6 - 8-8

I

- I/O errors
 - See linker error messages
- INCLUDE environment variable
 - See environment variables
- include files

- i option, 4-4
- environment variable, 4-5
- search order, 4-5
- searching for, 4-4
- initialized data, 8-2, 8-4
- input files, 2-2, 5-8
 - assembler, 4-2
 - source filename extensions, 2-2 - 2-3
- interrupt handlers
 - examples, 8-17
 - compile options, 8-17
 - defined, 8-16
 - int_end, 8-17
 - int_start, 8-17
 - preserving registers, 8-16 - 8-17
- interrupt service routines, 8-16

J

- jsr optimizations, 4-4
- jump table, segmentation, 8-6

K

- keywords
 - list, 3-2
 - extern, 8-15

L

- Language specs
 - See also data types
 - See also ANSI preprocessor features
 - ANSI syntax, 3-7
 - data type list, 3-3
 - escape sequences, 3-12
 - keywords list, 3-2
 - long character constants, 3-12
 - operator list, 3-2
 - preprocessor directive list, 3-2
 - register variables, 3-13
 - structure assignment, 3-11

structure passing, 3-11
symbol names, 3-13
LARGE CODE, 3-14
LARGE DATA, 3-14
lb utility, 5-3, 6-12 - 6-22
 adding modules, 6-18 - 6-19
 advanced features, 6-17
 basic features, 6-14
 creating a library, 6-14
 default extension, defining, 6-22
 deleting modules, 6-19 - 6-20
 extracting modules, 6-21
 function code options, 6-12
 getting arguments from a file, 6-16
 library argument, 6-12
 mod arguments, 6-13
 moving modules, 6-19
 naming modules, 6-15
 options argument, 6-12
 ord, 6-27
 qualifier options, 6-13
 reading arguments from a file, 6-13
 rebuilding a library, 6-22
 replacing modules, 6-20
libraries, 2-12, 2-35, 5-3, 5-9 - 5-10, 8-10
 examples, 5-5 - 5-6, 8-11
 adding modules, 6-18 - 6-19
 c.lib, 5-9, 8-10
 creating, 6-14
 deleting modules, 6-19 - 6-20
 extracting modules, 6-21
 lb utility, 6-17 - 6-22
 m.lib, 5-9
 moving modules, 6-19
 naming conventions, 8-11
 order of modules, 5-5 - 5-7, 6-15
 rebuilding, 6-22
 replacing modules, 6-20
line continuation, 2-44
linker

examples, 5-3 - 5-4, 5-8, 5-10
See also environment variables
CLI programs, 5-1
CLIB environment variable, 5-4, 5-11
device drivers, 5-1
executable file, 5-8
executable programs, 5-1
generates db symbol table, 7-3
hunk data, 5-14
input files, 5-8
introduction, 5-2
libraries, 5-3, 5-5, 5-9 - 5-10
relocatable object format, 5-1
starting, 5-7
syntax, 5-7
the process, 5-2 - 5-4
use, 5-7 - 5-8, 5-10
workbench programs, 5-1

linker error messages, 9-10
 command line summary list, 9-10 - 9-11
 explanations, command line errors, 9-70 - 9-71
 explanations, I/O errors, 9-71 - 9-73
 explanations, memory errors, 9-73
 explanations, source code errors, 9-74 - 9-76

linker options, 8-3
 +a long word boundaries, 5-10
 +c chip memory, 5-15, 8-3
 +f fast memory load, 5-10, 5-15, 8-3
 +l, 5-10
 +o, 5-11
 +s val, 5-11
 -a, 4-11
 -f, 5-10, 5-12, 8-10
 -g source level debug, 5-10, 5-16
 -l name, 5-10 - 5-11, 8-10
 -m, 5-10
 -o, 5-8, 5-11, 8-10
 -q, 5-11
 -t, 5-11
 -v, 5-11

-w, 5-11
listing file, assembler, 4-3
literals, wide, 3-10
loading programs, db, 7-3
loading symbols, db, 7-4
long character constant, 3-12
ls command, db, 7-29
ls utility, 6-23

M

m.lib, 8-11
m8.lib, 8-11
ma command, db, 7-29
ma.lib, 8-11
macro calls
 definition, 4-20
macros
 creating in db, 7-6
mb command, db, 7-30
mc command, db, 7-30
memory change breakpoints, db, 7-5
memory errors
 See linker error messages
memory models, 2-8 - 2-14
 assembler, large code, 4-5, 4-7
 assembler, large data, 4-5, 4-7
 code segmentation, 2-14, 8-4
 control of, 2-15
 large code vs small code, 2-9 - 2-11
 large data vs small data, 2-8 - 2-9
 libraries, 2-12
 multimodule programs, 2-13
 selection for use by a module, 2-12
 using the correct libraries, 8-11
mf command, db, 7-31
mf.lib, 8-11
mkarcv utility, 6-25
ml command, db, 7-30
mm command, db, 7-31
movem

- optimizations, 4-4
- reg directive, 4-19
- ms command, db, 7-31
- multi-tasking, 8-17
 - examples, 8-18
 - use of a4 register, 8-18
- multimodule programs, 2-13
- mw command, db, 7-30

N

- n command, db, 7-32
- naming conventions
 - C and assembly, 8-12
- numbering segments, 8-9
- numeric data
 - internal storage, 2-43

O

- obd utility, 6-26
- object code file
 - assembler, 4-3
 - creation, 2-3 - 2-4
- object modules, 8-11
- opcode error messages
 - See assembler error messages
- operating instructions
 - assembler, 4-2
- operator list, 3-2
- optimizations
 - assembler, 4-3 - 4-4
 - branches, 4-3
 - control, 2-15
 - jsr, 4-4
 - movem, 4-4
 - volatile, 3-5
- ord utility, 5-6, 6-27
- output control, 2-15
- output files, 2-3 - 2-4
- overlay breakpoints, db, 7-8

P

p command, db
 See print command, db

parser control, 2-15

pointer considerations, 2-41

pointers

- declare function arguments that are pointers, 2-42
- declare functions that return pointers, 2-41
- variables and constants as arguments, 2-42

portability, 3-1

- integers, 3-3

pragmas

- definition, 2-36
- a6 register, 2-36
- amicall, 2-36 - 2-40
- Amiga exec calls, 2-35
- calling resident libraries, 2-36 - 2-38
- creating resident libraries, 2-38 - 2-39
- libcall, 2-40
- regcall, 2-40
- register function calls, 2-39
- syscall, 2-40

pre-defined symbols, 3-13

precompiled #include files, 2-6

predefined symbols

- __FILE__, 3-14
- __FUNC__, 3-14
- __LINE__, 3-14
- __INT32, 3-14
- __LARGE_CODE, 3-14
- __LARGE_DATA, 3-14
- __AZTEC_C, 3-14
- __MCH_AMIGA, 3-14
- __MPU68000, 3-14
- __VERSION, 3-14

preprocessor

- concatenation, 3-15
- directives, 3-2
- features, 3-13

preserving registers, 8-15

See registers
print command, db, 7-2
 examples, 7-35 - 7-38
 count specifier, 7-33
 desc_codes, 7-33, 7-38
 format items, 7-33
 format string, 7-33
 indir, 7-34
 rpt parameter, 7-35
processor support
 assembler, 4-2
program control, db, 7-4
program organization, 8-2
program termination, db, 7-4, 7-8
prototypes, 2-28
 function call, 3-7
 function declarations, 3-6, 3-9
 function definition, 3-7

Q

q command, db, 7-39
QuikFix, 2-1 - 2-2, 2-19, 2-29, 2-34

R

r command, db, 7-40
RAM, 2-36
range
 definition, 7-13
register conventions
 assembler, 4-21
registers
 a4, 2-13, 2-26, 8-17
 a5, 2-30
 a6, 2-36
 control of, 2-15
 definition, 7-10
 floating point, 4-14
 preserving, 8-15 - 8-17
 reg directive, 4-19

- restoring, 4-22
- stack pointer, 4-21
- usage, 8-15
- variables, 3-13
- relocatable object format, 5-1
- requirements, db, 7-1
- reselecting segments, 8-10
- resident library
 - example, 2-39
 - Amiga functions, 2-37
 - calling, 2-36 - 2-37
 - creating, 2-38
- ROM, 2-36
- ROM applications
 - suppressing dseg generation, 2-26
- root segment
 - startup code, 8-7

S

- s command, db, 7-40
- s.lib, 8-11
- scatter loading support, db, 7-8
- segment loading and unloading, db, 7-9
- segmentation, 8-5
 - examples, 8-9 - 8-10
 - creating segments, 8-8
 - function exclusivity, 8-5
 - global data, 8-9
 - hunk data, 8-6, 8-8
 - jump table, 8-6
 - linker requirements, 8-9
 - linking startup code, 8-7
 - loading segments, 8-5
 - maximum number of segments, 8-9
 - memory usage, 8-8
 - numbering segments, 8-9
 - program design, 8-5
 - reselecting segments, 8-10
 - segload, 8-6
 - segment construction, 8-7

- segment data table, 8-8
- segment size, 8-4
- segment size limits, 8-4
- segmented code, 8-5, 8-8
- segments, 8-2 - 8-3
- static data, 8-9
- unloading segments, 8-6

set utility, 6-28

setdate utility, 6-29

setting and removing memory change breakpoints, db, 7-6

size types, 3-3

size_t, 3-13

sizeof, 3-4, 3-13

skip counter, db, 7-5

source code errors

- See linker error messages

source filename extensions, 2-2 - 2-3

source program structure

- comments, 4-8
- executable instructions, 4-8

special characters, db, 7-11

stack segments, 8-3

- C and assembly, 8-13
- CLI default value, 8-4
- function calls, 8-13
- stack size, 8-4

starting, db, 7-7

startup code, 8-7

- linking, 8-7
- location, 8-7
- root segment requirements, 8-7

static data, 8-9

stopping display, db, 7-9

stopping program execution, db, 7-9

string concatenation, 3-15

strings

- long, 3-15
- wide, 3-10

structure

- alignment, 3-4

- initialization, 3-10
- subroutines, arguments to
 - floating point, 4-21
 - returning from a C function, 4-22
 - variable names, 4-21
- subtasks, creating, 8-17
 - example, 8-18
- symbol names, 3-13
- symbol table hunks, db, 7-3
- syntax error messages
 - See assembler error messages

T

- table breakpoints, db, 7-5
- tables, 2-34
- TERM, definition, 7-10
- termination of program, db, 7-4
- toggle, 2-14
- trace mode, db, 7-6
- trigraphs, 2-28, 3-11
- turning off compiler options, 2-15

U

- u command, db, 7-41
- underscore
 - referencing symbols in db, 7-3
- uninitialized data, 8-3 - 8-4
- UNIX compatibility, 3-1
- unsigned characters, 3-4
- utilities:
 - adump - Amiga object module dump, 6-3
 - arcv - text file archiver, 6-2
 - cmp - file comparison, 6-4
 - cnm - display file info, 6-5 - 6-9
 - du - disk usage, 6-10
 - hd - hex dump, 6-11
 - lb - object file librarian, 6-12 - 6-22
 - ls - list files and directories, 6-23
 - mkarcv - text file archiver, 6-25

obd - list object code, 6-26
ord - sort object module, 6-27
set - set environment variable, 6-28
setdate - set date and time, 6-29

V

v command, db, 7-42
variables, 8-3
volatile, 3-5

W

warning control, 2-15
wide literals, 3-10
wide strings, 3-10
workbench, 8-19
 examples, 8-21
 creating programs, 8-19 - 8-22
 starting up, 8-21

X

x command, db, 7-42
XENIX compatibility, 3-1

Aztec C Library Manual for the Amiga

**Version 5.0
October 1989**

Copyright © 1988, 1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

PUBLISHED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

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

This software package and document are copyrighted ©1988, 1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

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, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh 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

October 1989	Fifth Edition
December 1988	Fourth Edition
May 1988	Third Edition
April 1986	Second Edition
February 1986	First Edition

Table of Contents

Chapter 1 - Introduction

ANSIC	1 - 1
-------------	-------

Chapter 2 - Library Overview

Introduction	2 - 1
Library Summary	2 - 1
31-Character Names	2 - 2
Integer Sizes	2 - 2
Overview of Aztec C and Amiga I/O	2 - 3
Aztec C I/O Functions	2 - 3
Preopened Devices and Command Line Arguments	2 - 4
File I/O	2 - 5
Sequential I/O	2 - 5
Random I/O	2 - 5
Opening Files	2 - 6
Device I/O	2 - 6
Console I/O	2 - 6
I/O to Other Devices	2 - 7
Mixing Unbuffered and Standard I/O Calls	2 - 7
AMIGA I/O FUNCTIONS	2 - 7
Preopened Devices and Command Line Arguments	2 - 8
FILE I/O	2 - 8
Device I/O	2 - 8
Aborting a Program from the Console	2 - 8
Overview of Aztec Standard I/O	2 - 9
Aztec C Standard I/O Functions	2 - 9
Opening Files and Devices	2 - 9

Closing Streams	2 - 10
Sequential I/O	2 - 10
Random I/O	2 - 10
Buffering	2 - 11
Errors	2 - 11
THE STANDARD I/O FUNCTIONS	2 - 12
AMIGA STANDARD I/O	2 - 14
Buffering	2 - 14
Overview of Aztec C and Amiga Unbuffered I/O	2 - 14
FILE I/O	2 - 16
DEVICE I/O	2 - 17
Unbuffered I/O to the Console	2 - 17
Unbuffered I/O to Nonconsole Devices	2 - 17
Overview of Aztec C and Amiga Console I/O	2 - 17
Character-Oriented Input	2 - 17
Writing System-Independent Programs	2 - 18
Using ioctl	2 - 18
THE sgtty FIELD	2 - 19
EXAMPLE	2 - 19
Console Input - RAW Mode	2 - 19
Overview of Aztec C and Amiga Dynamic Buffer Allocation	2 - 20
Dynamic Allocation of Standard I/O Buffers	2 - 21
Overview of Dynamic Buffer Allocation	2 - 21
Overview of Aztec C and Amiga Error Processing	2 - 21
Overview of ANSI Header Files	2 - 23
assert.h	2 - 23
ctype.h	2 - 24
errno.h	2 - 24
fcntl.h	2 - 24

float.h	2 - 24
limits.h	2 - 24
locale.h	2 - 24
math.h	2 - 24
setjmp.h	2 - 24
signal.h	2 - 24
stdarg.h	2 - 24
stddef.h	2 - 25
stdio.h	2 - 25
stdlib.h	2 - 25
string.h	2 - 25
time.h	2 - 25

Chapter 3 - Library Functions

Introduction	3 - 1
FUNCTION LIST	3 - 2
DESCRIPTION FORMAT	3 - 6
_abort (C) - Miscellaneous	3 - 9
_abs (C) - Integer Math	3 - 10
_access (C) - Standard I/O	3 - 11
_acos (M) - Float Math	3 - 13
_asctime (C) - Time	3 - 14
_asin (M) - Float Math	3 - 15
_assert (Macro) - Miscellaneous	3 - 16
_atan (M) - Float Math	3 - 17
_atan2 (M) - Float Math	3 - 18
_atexit (C) - Miscellaneous	3 - 19

atof (M) - Conversion	3 - 20
atoi(C) - Conversion	3 - 21
atol (C) - Conversion	3 - 22
calloc (C) - Memory Allocation	3 - 23
ceil (M) - Float Math	3 - 24
clearerr (C) - Standard I/O	3 - 25
_cli_parse (C) - Amiga	3 - 26
clock (C) - Time	3 - 27
close (C) - UNIX I/O	3 - 28
cos (M) - Float Math	3 - 29
cosh (M) - Float Math	3 - 30
cotan (M) - Float Math	3 - 31
creat (C) - UNIX I/O	3 - 32
ctime (C) - Time	3 - 33
difftime (C) - Time	3 - 34
div (C) - Integer Math	3 - 35
dos_packet (C) - Amiga	3 - 36
exec (C) - Amiga	3 - 37
Parameters	3 - 37
The Functions	3 - 38
Other Features	3 - 38
exit (C) - Miscellaneous	3 - 39
exp (M) - Float Math	3 - 40
fabs (M) - Float Math	3 - 41
fclose (C) - Standard I/O	3 - 42

fdopen (C) - UNIX I/O	3 - 43
feof (C) - Standard I/O	3 - 46
ferror (C) - Standard I/O	3 - 47
fexecl, fexecv(C) - Amiga	3 - 48
Parameters	3 - 48
The Functions	3 - 49
Other Information	3 - 49
fflush (C) - Standard I/O	3 - 51
fgetc (C) - Standard I/O	3 - 52
fgetpos (C) - Standard I/O	3 - 53
fgets (C) - Standard I/O	3 - 54
fileno (C) - UNIX I/O	3 - 55
floor (M) - Float Math	3 - 56
fmod (M) - Float Math	3 - 57
fopen (C) - Standard I/O	3 - 58
format (C) - Conversion	3 - 61
fprintf (C, M) - Standard I/O	3 - 62
fputc (C) - Standard I/O	3 - 63
fputs (C) - Standard I/O	3 - 64
fread (C) - Standard I/O	3 - 65
free (C) - Memory Allocation	3 - 66
freeseg (C) - Amiga	3 - 67
freopen (C) - Standard I/O	3 - 68
frexp (M) - Float Math	3 - 69
fscanf (C,M) - Standard I/O	3 - 70

fseek (C) - Standard I/O	3 - 71
fsetpos (C) - Standard I/O	3 - 73
ftell (C) - Standard I/O	3 - 74
ftoa (M) - Conversion	3 - 75
fwrite (C) - Standard I/O	3 - 76
geta4 (C) - Amiga	3 - 78
getc (C) - Standard I/O	3 - 79
getchar (C) - Standard I/O	3 - 80
getenv (C) - Miscellaneous	3 - 81
gets (C) - Standard I/O	3 - 82
gmtime (C) - Time	3 - 83
index (C) - String Handling	3 - 84
int_end (C) - Amiga	3 - 85
int_start (C) - Amiga	3 - 86
ioctl (C) - UNIX I/O	3 - 87
is... (C) - Character Type	3 - 88
labs (C) - Integer Math	3 - 90
ldexp(M) - Float Math	3 - 91
ldiv (C) - Integer Math	3 - 92
localtime (C) - Time	3 - 93
log (M) - Float Math	3 - 94
log10 (M) - Float Math	3 - 95
longjmp (C) - Miscellaneous	3 - 96
lseek (C) - UNIX I/O	3 - 97
malloc (C) - Memory Allocation	3 - 99

memchr (C) - Block Operation	3 - 100
memcmp (C) - Block Operation	3 - 101
memcpy (C) - Block Operation	3 - 102
memmove (C) Block Operation	3 - 103
memset (C) - Block Operation	3 - 104
mktemp (C) - UNIX I/O	3 - 105
mkttime (C) - Time	3 - 107
modf (M) - Float Math	3 - 108
movmem(C) Block Operation	3 - 109
open (C) - UNIX I/O	3 - 110
perror (C) - Standard I/O	3 - 113
pow (M) - Float Math	3 - 115
printf (C, M) - Standard I/O	3 - 116
The Format String	3 - 116
Conversion Specifiers	3 - 116
Flags Field	3 - 117
Width Field	3 - 118
Precision Field	3 - 118
Size-mod Field	3 - 118
Type Field	3 - 119
putc (C) - Standard I/O	3 - 122
putchar (C) - Standard I/O	3 - 123
puts (C) - Standard I/O	3 - 124
qsort (C) - Miscellaneous	3 - 125
ran (M) - Float Math	3 - 127
rand (C) - Integer Math	3 - 128

read (C) - UNIX I/O	3 - 129
realloc (C) - Memory Allocation	3 - 130
remove (C) - Standard I/O	3 - 131
rename (C) - Standard I/O	3 - 132
rewind (C) - Standard I/O	3 - 133
rindex (C) - String Handling	3 - 134
sbrk (C) - Memory Allocation	3 - 135
scanf (C,M) - Standard I/O	3 - 136
THE FORMAT STRING	3 - 136
Matching White Space Characters	3 - 136
Matching Ordinary Characters	3 - 137
Matching Conversion Specifications	3 - 137
Details of Input Conversion	3 - 137
scdir (C) - Amiga	3 - 141
screen (S) - Amiga	3 - 142
segload (C) - Amiga	3 - 144
setbuf (C) - Standard I/O	3 - 145
setenv (C) - Miscellaneous	3 - 146
setjmp - Miscellaneous	3 - 147
setvbuf (C) - Standard I/O	3 - 148
signal (C) - Signal	3 - 150
Signals and the sig Parameter	3 - 150
Signal Processing and the func Parameter	3 - 150
Return Values from Signal	3 - 151
sin (M) - Float Math	3 - 153
sinh (M) - Float Math	3 - 154

sprintf (C, M) - Conversion	3 - 155
sqrt (M) - Float Math	3 - 156
srand (M) - Float Math	3 - 157
strrand (C) - Integer Math	3 - 158
sscanf (C,M) - Conversion	3 - 159
stat (C) - UNIX I/O	3 - 160
_stkchk (C) - Miscellaneous	3 - 161
_stkover (C) - Miscellaneous	3 - 162
strcat (C) - String Handling	3 - 163
strchr (C) - String Handling	3 - 164
strcmp (C) - String Handling	3 - 165
strcoll (C) - String Handling	3 - 167
strcpy (C) - String Handling	3 - 168
strcspn (C) - String Handling	3 - 169
strlen (C) - String Handling	3 - 170
strncat (C) - String Handling	3 - 171
strncmp (C) - String Handling	3 - 172
strncpy (C) - String Handling	3 - 173
strpbrk (C) - String Handling	3 - 174
strrchr (C) - String Handling	3 - 175
strspn (C) - String Handling	3 - 176
strstr (C) - String Handling	3 - 177
strtod (C) - Conversion	3 - 178
strtok (C) - String Handling	3 - 179
strxfrm (C) - String Handling	3 - 181

tan (M) - Float Math	3 - 182
tanh (M) - Float Math	3 - 183
time (C) - Time	3 - 184
tmpfile (C) - Standard I/O	3 - 185
tmpnam (C) - Standard I/O	3 - 186
tolower (C) - Conversion	3 - 187
toupper (C) - Conversion	3 - 188
ungetc (C) - Standard I/O	3 - 189
unlink (C) - UNIX I/O	3 - 190
va_arg, va_end, va_start (C) - Variable Arguments	3 - 191
vfprintf (C) - Standard I/O	3 - 192
vprintf (C) - Standard I/O	3 - 193
vsprintf (C) - Conversion	3 - 194
_wb_parse (C) - Amiga	3 - 195
write (C) - UNIX I/O	3 - 196

Chapter 4 - Workbench

Amiga Reference Guides	4 - 1
Amiga Workbench Functions	4 - 2

INTRODUCTION

LIBRARY OVERVIEW

LIBRARY FUNCTIONS

WORKBENCH

INTRODUCTION

1

Chapter 1 - Introduction

This is the *Aztec C Library Manual* and is included as a part of your total Aztec C documentation package. See your *Aztec C User Guide* for a listing of all of the Aztec C configurations and documentation that are available for the Amiga.

The *Aztec C User Guide* showed you how to install your software and how to begin using Aztec C easily and quickly. The *Aztec C Reference Manual* presented more advanced programs and a more thorough understanding of how the Aztec C package works.

This *Aztec C Library Manual* includes an overview, listing, and description of all the library functions that are included with your Aztec C package, and a listing of the Amiga Workbench function names and syntax.

Of course, this *Aztec C Library Manual* also contains a detailed Table of Contents and an Index.

ANSI C

Aztec C is fully dpANSI standard. If you have used Aztec C prior to Version 5.0, you should note that this standardization has caused the library functions included with your Aztec C system to change substantially. Before using Aztec C, you should carefully review the Library Overview chapter and the Library Functions chapter for a more complete understanding of ANSI and how it has affected the library functions.

LIBRARY OVERVIEW

2

Chapter 2 - Library Overview

Introduction

This chapter discusses the libraries included with Aztec C for the Amiga and gives an overview, by category, of the Aztec and Amiga functions provided in this manual.

The chapter first presents a library summary and then divides into the following sections.

1. Overview of Aztec C and Amiga I/O
2. Overview of Aztec C Standard I/O
3. Overview of Aztec C and Amiga Unbuffered I/O
4. Overview of Aztec C and Amiga Console I/O
5. Overview of Aztec C and Amiga Dynamic Buffer Allocation
6. Overview of Aztec C and Amiga Error Processing
7. Overview of ANSI Header Files

Library Summary

More detailed information about Aztec C libraries is available in the **Getting Started** Chapter of the *Aztec C User Guide*. If you are planning to

use special libraries for 16 bit **ints** or specialized floating point libraries, you should read the material in that chapter.

If you are using Amiga Workbench routines, you should read material relating to the Amiga Workbench functions, the **functions.h** header, and direct resident library calls, in the **Getting Started Chapter** of the *Aztec C User Guide* and the **Compiler Chapter** of the *Aztec C Reference Manual*.

Aztec C for the Amiga features support for Workbench functions and includes the following libraries:

c.lib	standard I/O and Amiga toolbox routines
m.lib	Manx IEEE double and extended precision math library
mf.lib	Motorola Fast Floating Point math library
ma.lib	Amiga IEEE double precision math library
m8.lib	68881 coprocessor math library
s.lib	screen functions

The distributed headers and libraries reflect the following:

31-Character Names

Global names in Aztec C have 31 significant characters and have a leading underscore.

Integer Sizes

c.lib uses and expects 32 bit integers. If source files are compiled using 16 bit integers (such as with the **-ps** option) then the 16 bit libraries such as **c16.lib** must be used when linking. Additional information on special libraries and alternate data definitions can be found in the **Getting Started Chapter** of the *Aztec C User Guide*.

Note to previous users: Prior to Version 5.0, Aztec C's default integer size was 16 bits.

Overview of Aztec C and Amiga 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 Chapter 8 of *The C Programming Language*.

Aztec C I/O Functions

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 that perform their own blocking and unblocking 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 unblocking the file records.

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," e.g., 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 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 that performed the open and must be closed by the appropriate function in the same set. There are exceptions to this nonintermingling that 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.

Preopened Devices and Command Line Arguments

There are three logical devices which are automatically opened when a program is started: standard input, standard output, and standard error. By default, these are associated with the console. The operator, as part of the command line which starts the program, can specify that these logical devices are to be "redirected" to another device or file. Standard input is redirected by entering on the command line, after the program name, the name of the file or device, preceded by the character <. Standard output is redirected by entering the name of the file or device, preceded by >.

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 < exmplfil
```

And this will copy **exmplfil** to **testfile**:

```
    cpy < exmplfil > testfile
```

Aztec C will pass command line arguments to your program via the function **main(argc, argv)**. **argc** is an integer containing the number of arguments plus one; **argv** is a pointer to 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 your function **main()**. The first parameter to main is the integer 4. The second parameter is a pointer to an array of four character 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 your program and I/O redirection specifications. The I/O redirection strings will not be passed to your program and can appear anywhere on the command line after the command name but BEFORE any arguments. For example, the standard output of the **prog** program can be redirected to the file **outfile** by the following command; in this case the argc and argv parameters to the main function of **prog** are the same as if the redirection specifier was not present:

```
prog > outfile arg1 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 simply 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 that 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.

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 filenames and the areas searched for files differ from system to system.

Device I/O

Aztec C allows programs to access devices as well as files. Each system has its own names for devices.

Console I/O

Console I/O can be performed in a variety of ways. There is a default mode, and other modes can be selected by calling the function ioctl().

When the console is in default mode, console input is buffered and is read from the keyboard one line at a time. Typed characters are echoed to the screen and you 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 its 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
- to terminate a read request if a key is not depressed within a certain interval.

Output to the console is always line buffered: Display occurs when a newline is output or when a request for console input is made. The only choice concerns translation of the newline character; by default, this is translated into a carriage return, line feed sequence.

Optionally, 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. The options available on the Amiga are discussed in the *Amiga Rom Kernel Reference Manual*.

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.

AMIGA I/O FUNCTIONS

A maximum of 20 files and devices, including the standard I/O devices, can be opened at once for both standard and unbuffered I/O. When this limit is reached, an open file or device must be closed before another can be opened.

Amiga functions are also supplied for accessing files and devices--a program can access some files and devices using the Amiga functions and others using the ANSI-compatible functions.

The ANSI-compatible I/O functions contain tables having a fixed number of entries, one for each open file or device. The number of entries in the table sets the limit on the number of files and devices that can be accessed simultaneously using the ANSI-compatible functions. AmigaDos has similar limits on the number of files and devices that can be accessed. Using an AmigaDos entry will simultaneously tie up an ANSI entry. There are more available AmigaDos entries than ANSI entries. If the ANSI table is full, you might choose to open a device or file using AmigaDos I/O functions.

Preopened Devices and Command Line Arguments

For programs running on an Amiga, the program name is pointed at by the first item in the array that is pointed at by the second argument of the program `main()` function. That is, if the `main()` function begins

```
main(argc, argv)
int argc; char *argv[];
```

the `argv[0]` is a pointer to the program name.

FILE I/O

Unlike some systems supported by Aztec C, positioning of a file relative to its end is always correctly done with the following limitation: You cannot position a file beyond its end.

Device I/O

Using the ANSI-compatible I/O functions, a program can access devices using their AmigaDOS names. For a list of these names, and a description of device I/O using AmigaDOS, see the *AmigaDOS* manuals.

A program can also access devices using Amiga function calls. These calls bypass the ANSI-compatible I/O system and AmigaDOS, and give a program the most control over devices. For example, when accessing the serial device using the Amiga functions, a program has control over the device's baud rate, etc., which it does not have when accessing the serial device using the ANSI-compatible or AmigaDOS functions.

The ANSI-compatible functions supplied with Aztec C support console I/O differently on other systems than they do on the Amiga. On other systems, there is one console device and a program can issue calls to the `ioctl()` function to define how console I/O is to be performed. On the Amiga, `ioctl()` is only supported for setting RAW mode. See the `ioctl()` command in the **Library Functions** chapter of this manual for more information.

Aborting a Program from the Console

To abort a program, the operator types `^C` which `Chk_Abort()` detects. `Chk_Abort()` calls a routine called `_abort()` which prints a `^C` and then calls `exit()`. This routine can be replaced by the program so that the program can do any cleaning up that is necessary before exiting.

When called, `Chk_Abort()` aborts the program if (1) ^C is typed and (2) the global long `Enable_Abort()` is nonzero (which it is by default). If either of these conditions is not satisfied, `Chk_Abort()` simply returns.

Therefore, to disable the ability of the operator to abort a program by typing ^C, a program must set `Enable_Abort()` to 0. `Chk_Abort()` is called in all the low level I/O routines such as `open()`, `read()`, `write()`, and `Iseek()`.

Overview of Aztec Standard I/O

Aztec C Standard I/O Functions

The standard I/O functions are used by programs to access files and devices. Aztec C functions are fully compatible with the ANSI standard.

These functions provide programs with convenient and efficient access to files and devices. When accessing files, the functions buffer the file data; that is, they handle the blocking and unblocking of file data. Therefore, your program can concentrate on its own concerns.

Opening Files and Devices

Before a program can access a file or device, it must be "opened," and when processing on a file or device 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.

It contains such information as the file position, pointer to an associated buffer, an error indicator, and an end-of-file indicator.

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()` 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:

- 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 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 could not be done. If the program lets the system automatically close the file, the program will not 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.

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.

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.

Your program specifies a buffer to be used for a file by calling the function `setbuf()` 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, you have not 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 its buffer will be freed, by calling `free()`.

Dynamically allocated buffers are obtained from a particular region of memory (the heap), whether requested by the standard I/O functions or by your program.

The default size of an I/O buffer is given by the manifest constant `BUFSIZ` in `stdio.h`.

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 `setbuf()` function.

Errors

There are three fields that may be set when an exceptional condition occurs during stream I/O. Two of the fields are unique to each stream (i.e., 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 (EOF) 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 EOF 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 EOF is reached on an input stream, a standard I/O function returns EOF after setting the stream's EOF 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.

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. Thus `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, its 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. Functions marked obsolete will not be supported in future releases. You will also need to refer to release 3.6 documentation for detailed specifications for some of these functions.

Here are the standard I/O functions that can only access `stdin`, `stdout`, and `stderr`. These are all ASCII functions; i.e., they expect to deal with text characters only.

<code>getchar()</code>	Get an ASCII character from <code>stdin</code>
<code>gets()</code>	Get a line of ASCII characters from <code>stdin</code>
<code>printf()</code>	Format data and send it to <code>stdout</code>
<code>*puterr()</code>	Send a character to <code>stderr</code> - obsolete
<code>putchar()</code>	Send a character to <code>stdout</code>
<code>puts()</code>	Send a character string to <code>stdout</code>
<code>scanf()</code>	Get a line from <code>stdin</code> and convert it
<code>vprintf()</code>	Format data using a variable argument list and send it to <code>stdout</code> .

Here are the rest of the standard I/O functions. Those preceded by an asterisk (*) are non-ANSI functions:

<code>clearerr()</code>	Clear EOF and error flag for stream
<code>fclose()</code>	Close an open stream
<code>*fdopen()</code>	Open as a stream a file or device already open for unbuffered I/O
<code>feof()</code>	Check for EOF on a stream
<code>ferror()</code>	Check for error on a stream
<code>fflush()</code>	Write stream's buffer
<code>fgetc()</code>	Get the next character from an input stream
<code>fgetpos()</code>	Get information on file position
<code>fgets()</code>	Get a line of ASCII characters
<code>*fileno()</code>	Get file descriptor associated with stream
<code>fopen()</code>	Open a file or device
<code>fprintf()</code>	Format data and write it to a stream
<code>fputc()</code>	Write a character to an output stream
<code>fputs()</code>	Send a string of ASCII characters to a stream
<code>fread()</code>	Read data from a specified stream.
<code>freopen()</code>	Open an open stream to another file or device
<code>fscanf()</code>	Get data and convert it

fseek()	Set current position within a file
fsetpos()	Set file position
ftell()	Get current position
fwrite()	Write data to a stream
getc()	Macro for fgetc()
*getw()	Get two characters from stream - obsolete
perror	Map errno to a descriptive string
putc	Macro for fputc()
*putw	Send two characters to a stream - obsolete
remove	Deletes a file
rename	Changes the name by which a file is accessed
rewind	Set file position to
setbuf	Specify buffer and buffering
setvbuf	Specify buffer, buffering, and buffer size
tmpfile	Creates a temporary binary file
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

AMIGA STANDARD I/O

Buffering

On the Amiga, the size of a buffer used for standard I/O is 1024 bytes.

Overview of Aztec C and Amiga Unbuffered I/O

The Aztec C unbuffered I/O functions are used to access files and devices. They are similar to their UNIX counterparts with the addition that a file may be opened as either text or binary. The Amiga unbuffered I/O functions are described at the end of this section.

A program using these functions 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 buffered, unless specific actions are taken by your program.

Programs which use the unbuffered I/O functions to access files generally handle the blocking and unblocking of file data themselves. Programs requiring file access but unwilling to perform the blocking and unblocking can use the standard I/O functions.

Here are the unbuffered I/O functions:

access()	Determine if a file can be accessed in a specified manner.
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
rename()	Rename a file
unlink()	Delete a file
write()	Write data to an open file or device

The header file **fcntl.h** is designed to be used in conjunction with the unbuffered I/O routines. It includes function declarations, macros to be used with these functions, and the **_dev** structure declaration.

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 open for unbuffered I/O. First, you 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, you can automatically open the file or device 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.

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**.

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 has been transferred.

The current position of a file can be manipulated in various ways by a program, allowing both sequential and random access 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.

open() provides a mode, **O_APPEND**, which causes the file being opened to be positioned at its end. 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.

DEVICE I/O

Unbuffered I/O to the Console

There are several options available when accessing the console. Here we 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 Nonconsole Devices

Unbuffered I/O to devices other than the console is truly unbuffered.

Overview of Aztec C and Amiga Console I/O

A program has control over several options relating to console I/O. The primary option supported by the Amiga is the RAW mode, described below.

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.

The type of character-oriented console input which the Amiga supports is type RAW. In RAW mode, a program does not have control over the other console options.

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.

The standard I/O functions all read 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 options 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 selects console I/O modes using the function `ioctl()`. This has the form:

```
#include <sgtty.h>
ioctl(fd, code, arg)
struct sgttyb *arg;
```

The header file `sgtty.h` defines symbolic values for the code parameter (which tells `ioctl` what to do) and the structure `sgttyb`.

The parameter `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. Here, `ioctl` always applies to the console.

The parameter `code` defines the action to be performed by `ioctl()`. It can have these 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*.

The argument **arg** points to a structure named **sgttyb** that contains the following field:

```
int sg_flags;
```

To set console options, a program should fetch the current state of the fields, using **ioctl()**'s TIOCGETP option. Then it should modify the fields to the appropriate values and call **ioctl()** again, using the **ioctl()** TIOCSETP option.

THE **sgtty** FIELD

sg_flags contains the UNIX-compatible flag **RAW**, which sets RAW mode and turns off other options. By default, **RAW** is disabled.

On some systems, other flags are contained in **sg_flags**.

More than one flag can be specified in a single call to **ioctl()**; the values are simply ORed 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** is reset, the console is set in line-oriented input mode.

EXAMPLE

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 will not 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:x/y/width/
        height/", "r")) == NULL){
        printf("can't open the console\n");
        exit();
    }
    ioctl(fileno(fp), TIOCGETP, &stty);
    stty.sg_flags |= RAW;
    ioctl(fileno(fp), TIOCSETP, &stty);
    for ( ; ){
        c = getc(fp);
        . . .
    }
}
```

Overview of Aztec C and Amiga 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:

<code>malloc()</code>	Allocate a buffer
<code>calloc()</code>	Allocate a buffer and initialize it to zeroes
<code>realloc()</code>	Allocate more space to a previously allocated buffer
<code>free()</code>	Release an allocated buffer for reuse

These functions are described in detail in the Library Functions chapter of this manual.

In addition, on some systems the UNIX-compatible functions `sbrk()` and `brk()` provide a more elementary means to allocate heap space. The malloc-type functions call `sbrk()` to get heap space, which they then manage.

On some systems, non-UNIX memory allocation functions are also supported.

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 your program. Standard I/O calls to dynamically allocate and deallocate buffers can be interspersed with those of your program.

Programs that perform standard I/O and have absolute control of the heap can explicitly define the buffers to be used by a standard I/O stream.

Overview of Dynamic Buffer Allocation

The Amiga function `lmalloc()` is a non-UNIX memory allocation function. It is obsolete. It is supported in this release for compatibility. There is no detailed information for `lmalloc()` in this document.

Note: For more information about specific functions, see the Library Functions chapter. For more information about the heap, see the "Program Organization" section of the Technical Information chapter of your *Aztec C Reference Manual*.

Overview of Aztec C and Amiga 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 floating point math (e.g., `sin()`, `sqrt()`) functions as part of their error processing.

When a standard I/O, unbuffered I/O, or math function detects an error, it sets a code in `errno` which describes the error. If no error occurs, the math 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 dependent on the type of error that occurred and sets **errno**. For example, an overflow would return **HUGE_VAL**.

When performing math 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 nonzero, 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 math function calls.

Since **errno** may be modified by an I/O function even if an error did not 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.

The following table lists the system-independent values that 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:

<i>Error code</i>	<i>Meaning</i>
ENOENT	File or directory does not exist
E2BIG	Argument list too long
EBADF	Bad file descriptor - file is not open or improper operation requested
ENOMEM	Insufficient memory for requested operation
EEXIST	File already exists on <code>creat()</code> request
EINVAL	Invalid argument
ENFILE	Exceeded maximum number of open files
EMFILE	Exceeded maximum number of file descriptors
ENOTTY	<code>ioctl()</code> attempted on non-console device
EACCES	Invalid access request, permission denied
EIO	I/O error (usually physical)
ENOSPC	No space left on device
ERANGE	Math function result too large
EDOM	Domain of argument to math function invalid
ENOEXEC	<code>fexec()</code> format error
EROFS	Read-only file system
EXDEV	Cross-device rename
EAGAIN	Nothing to read

Overview of 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 may be used to display the file name and line number for a call that fails.

ctype.h

Includes several function declarations and macros useful for testing and mapping characters.

errno.h

Includes macros for the reporting of error conditions. See the section **Overview of Aztec C and Amiga Error Processing**.

fcntl.h

Includes function declarations and macros for use with unbuffered I/O.

float.h

Includes macros that define various floating point sizes and limits.

limits.h

Includes macros that define the numerical limits for various ordinal types.

locale.h

Includes macros and function declarations supporting the "C" locale necessary for C translation.

math.h

Includes declarations for several mathematical functions that take double-precision arguments and return double-precision values.

setjmp.h

Includes the **setjmp()** macro for use with the **longjmp()** function.

signal.h

Includes function declarations and macros for handling various signals.

stdarg.h

Includes macros for advancing through a variable length argument list.

stddef.h

Includes type definitions for pointer subtraction, result of `sizeof` operator (which is now 32 bits), and largest available character set.

stdio.h

Includes types, macros, and function declarations for buffered I/O.

stdlib.h

Includes types and function declarations for string conversion, memory management, and other general utilities.

string.h

Includes function declarations for manipulating arrays of character type.

time.h

Includes types and function declarations for obtaining the current date and time as well as manipulating it and other time structures.



Chapter 3 - Library Functions

Introduction

This chapter contains two categories of library functions:

- System-independent functions that are common to all Aztec C packages.
- Machine-dependent functions that work only on the Amiga machines.

Note: The Amiga WorkBench routines are also discussed in their own chapter.

To help you access the information more easily and efficiently:

- Each function description is listed alphabetically
- Each function description begins on a new page.

This introduction includes a list of all included library functions as well as an explanation of the format that the function descriptions feature.

The individual function descriptions begin on page 3-9.

FUNCTION LIST

The functions described in this chapter include:

_abort() - abort task
abs() - return the absolute value of a signed integer
access() - determine accessibility of a file or directory
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() - indicate a 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
calloc() - allocate space for an array of objects from system memory
ceil() - compute the smallest integer not less than a specified number
clearerr() - clear end of file and error conditions in a stream
_cli_parse() - parse command line
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() - returns the cotangent of a double value
creat() - create a new file
ctime() - convert a time value to an ASCII string
difftime() - compute the difference between two times
div() - compute the quotient and remainder of the division of two integers
dos_packet() - output packet
exec1(), execv(), execlp(), execvp() - load and start another program
exit(), _exit() - terminate calling program
exp() - compute the exponential function e^x

fabs() - return the absolute value of a given number
fclose() - close a buffered I/O stream
fdopen() - open a file or device previously opened for buffered I/O
 for standard I/O
feof() - test for end-of-file in a standard I/O stream
ferror() - test for an error in a standard I/O stream
fexecl(), fexecv() - load and start another program
fflush() - flush an I/O stream
fgetc() - return the next available character from a standard I/O stream
fgetpos() - save the current file position for a stream
fgets() - get a string of characters from a stream
fileno() - return file descriptor associated with a stream
floor() - compute the largest integer not greater than a specified
 number.
fmod() - return the remainder of the double value x/y
fopen() - open a specified file or device for standard I/O access
format() - write formatted data using a user-defined output function
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
freeseg() - segment unload function
freopen() - reopen a stream with a new device
frexp() - decompose a floating point number
fscanf() - perform formatted input conversion on a specified stream
fseek() - reposition current location within a stream
fsetpos() - set the correct file position for a stream
ftell() - return the current file position within a standard I/O stream
ftoa() - convert floating point number to an ASCII string
fwrite() - write to a specified standard I/O stream
geta4() - set up a4 register
getc() - return the next available character from a standard I/O stream
getchar() - return the next available character from `stdin`
getenv() - get value of environment variable

gets() - get a string of characters from `stdin`
gmtime() - convert a calendar time into coordinated universal time
index() - find the first occurrence of a character in a string
int_end() - restore machine state from within a C interrupt handler
int_start() - set up a function as an interrupt handler
ioctl() - device I/O utility
labs() - return the absolute value of a signed long
ldexp() - build real numbers
ldiv() - compute quotient and remainder of the division of two longs
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
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
mktemp() - make a unique file name
mktimed() - convert a time value between formats
modf() - break a floating point value into integral and fractional parts
movmem() - copy a block of memory
open() - open device or file for unbuffered I/O
perror() - print a system error message
pow() - compute x to the y th power
printf() - formatted output function
putc() - write a character to an I/O stream
putchar() - write a character to the `stdout` stream
puts() - write a string to `stdout`
qsort() - sort an array of records in memory
ran() - generate floating point random numbers
rand() - return a pseudo-random integer
read() - read from a device or file using unbuffered I/O

realloc() - re-allocate an already existing memory block to a different size
remove() - delete a file
rename() - rename a disk file
rewind() - reposition a stream's position indicator to the beginning of the file
rindex() - find the last occurrence of a character in a string
sbrk() - memory allocation function
scanf() - perform formatted input conversion on the **stdin** stream
scdir() - return the name of the next file matching pattern
segload() - load a program segment
setbuf() - associate an I/O stream with a specific buffer
setenv() - set environment variables
setjmp() - set up for a non-local **goto**
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 the non-negative square root of a value.
srand() - set the random number seed for **ran()**
rand() - initialize the seed value used by **rand()**
sscanf() - perform formatted input conversion on a buffer
stat() - return file information
_stkchk() - check for stack overflow
_stkover() - report stack overflow
strcat() - concatenate two strings together
 strchr() - search for the first occurrence of a character in a string
strcmp() - compare two strings
strcoll() - compare two strings using the current locale
strcpy() - copy one string to another
strcspn() - return the index of the first character in a string *str1* which is contained in a string *str2*
strlen() - return the length of a string

strncat() - concatenate two strings together, up to *max* characters
strcmp() - compare two strings, up to *max* characters.
strcpy() - copy up to *max* characters from one string to another
strpbrk() - return the first occurrence in a string *str1* of a character in
 a string *str2*.
strrchr() - search for the last occurrence of a character in a string
strspn() - return the index of the first character in a string *str1* which
 is not contained in a string *str2*
strstr() - return a pointer of the first occurrence of a string *str2* within
 a string *str1*
strtod() - convert a string to a double
strtok() - tokenize a string
strxfrm() - transform a string to match the current locale
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_arg(), **va_end()**, **va_start()** - 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
_wb_parse() - open standard I/O window
write() - write to a file or device using unbuffered I/O

DESCRIPTION FORMAT

The description format is as shown on the following pages.

TYPE Lists the name of the function, one or more letters in parentheses which specify the libraries that contain the function, and the function category. The letters which may appear in parentheses and their corresponding libraries are:

C	c.lib, c16.lib, cl.lib, cl16.lib
M	m.lib, m8.lib, mf.lib, ma.lib
S	s.lib

At the right margin, in italics, is the standard to which the function adheres, (i.e., ANSI, UNIX, etc.)

NAME Lists the name of the function and a brief description of its use.

SYNOPSIS Indicates the types of arguments that the functions require, and the values it returns. For example, the function `atof` converts character strings into double precision numbers. It is listed in the synopsis as

`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:

`double atof(const char *cp);`

Unless otherwise noted, the function's declaration is contained in the header file specified for that function.

The notation

`#include <stdlib.h>`

at the beginning of a synopsis indicates that such a statement should appear at the beginning of any program calling that function.

DESCRIPTION Describes the function.

SEE ALSO Lists other relevant functions. You should also refer to the **Library Overview** chapter for further information about many of the functions discussed in this chapter.

DIAGNOSTICS Describes the error codes that the function may return. The **Errors Section** in the **Library Overview** chapter presents an overview of error processing.

EXAMPLES Gives examples on use of the function.

TYPE	<code>_abort (C) - Miscellaneous</code>	<i>Amiga</i>
NAME	<code>_abort</code> - abort task	
SYNOPSIS	<code>void _abort(void);</code>	
DESCRIPTION	<p><code>_abort()</code> is a function that is called by the function <code>Chk_Abort()</code> if the flag <code>Enable-Abort</code> is nonzero and the <code>^C</code> key is entered by the user (or if the <code>^C</code> signal is sent to the task in general.) The library version of <code>_abort()</code> displays <code>^C</code> on the screen and calls <code>exit(1)</code>.</p> <p>You may provide your own <code>_abort()</code> routine to take special action if <code>^C</code> is pressed. For example, the compiler, assembler, and linker remove any temporary files when <code>^C</code> is typed. <code>_abort()</code> calls a cleanup function that deletes the files, prints a message, and then calls <code>exit()</code>.</p>	

TYPE	abs (C) - Integer Math	<i>ANSI</i>
NAME	abs - return the absolute value of a signed integer	
SYNOPSIS	#include <stdlib.h> int abs (int x);	
DESCRIPTION	abs() computes and returns the absolute value of the signed integer <i>x</i> . 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()	

TYPE	access (C) - Standard I/O	Aztec										
NAME	access - determine accessibility of a file or directory											
SYNOPSIS	<pre>#include <fcntl.h> int access (char *filename, int mode);</pre>											
DESCRIPTION	<p>access() determines whether a file or directory can be accessed in a specified manner.</p> <p>The parameter <i>filename</i> points to the name of the file or directory; this name optionally contains the drive and path of the directories that must be passed through to get to the file or directory. If the drive component is not specified, the file or directory is assumed to reside on the default drive. If the path component is not specified, the file or directory is assumed to reside in the current directory on the specified drive.</p> <p><i>mode</i> is an int that specifies the type of access desired:</p> <table><thead><tr><th><i>Mode</i></th><th><i>Meaning</i></th></tr></thead><tbody><tr><td>4</td><td>read</td></tr><tr><td>2</td><td>write</td></tr><tr><td>1</td><td>execute (if a file) or search (if a directory)</td></tr><tr><td>0</td><td>existence of the file or directory.</td></tr></tbody></table> <p>If the existence of the file or directory is being checked (i.e., <i>mode</i> =0), access() returns 0 if the file exists and -1 if it does not. In the latter case, access() also sets the symbolic value ENOENT in the global integer <i>errno</i>.</p> <p>When access() is called to determine if a file can be accessed in a certain way (i.e., <i>mode</i> is not 0), access() returns 0 if the file can be accessed in the desired manner; otherwise, it returns -1 and sets a code in the global integer <i>errno</i> that defines why the access is not permitted.</p>	<i>Mode</i>	<i>Meaning</i>	4	read	2	write	1	execute (if a file) or search (if a directory)	0	existence of the file or directory.	LIBRARY FUNCTIONS
<i>Mode</i>	<i>Meaning</i>											
4	read											
2	write											
1	execute (if a file) or search (if a directory)											
0	existence of the file or directory.											

When read or write privileges are requested on a directory, **access()** reports on whether files can be read or written to within the directory, not whether or not the directory itself may be read or written to.

The symbolic values that **access()** may set in **errno** when it is called with a non-zero *mode* parameter are:

<i>errno</i>	<i>Meaning</i>
ENOTDIR	A component of the path prefix is not a directory.
ENOENT	The file or directory does not exist.
EACCES	File or directory cannot be accessed in the desired manner

TYPE	acos (M) - Float Math	<i>ANSI</i>
NAME	acos - return the arc cosine of a double value	
SYNOPSIS	<pre>#include <math.h> double acos (double x);</pre>	
DESCRIPTION	<p>acos() returns the arc cosine of <i>x</i>. <i>x</i> should be specified in radians. If <i>x</i> is outside of the range -1 to 1, then acos() will return 0 and set errno equal to EDOM.</p>	

TYPE	asctime (C) - Time	ANSI
NAME	asctime - translate a time value into an ASCII string	
SYNOPSIS	#include <time.h> char *asctime (const struct tm * <i>timeptr</i>);	
DESCRIPTION	asctime() creates an ASCII text string corresponding to the time contained in <i>timeptr</i> . The general format of the resulting string is: <code>Mon Apr 9 08:29:00 1968\n\o</code>	
	A pointer to the text string is returned by asctime.	
SEE ALSO	clock(), ctime(), difftime(), localtime(), time()	

TYPE	asin (M) - Float Math	<i>ANSI</i>
NAME	asin - return the arc sine of a double value	
SYNOPSIS	<pre>#include <math.h> double asin(double x);</pre>	
DESCRIPTION	<p>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.</p>	

TYPE	assert (Macro) - Miscellaneous	ANSI
NAME	assert - verify program assertion	
SYNOPSIS	#include <assert.h> void assert (int <i>expression</i>);	
DESCRIPTION	The assert() macro is useful for putting diagnostic messages in a program. assert() determines whether <i>expression</i> is true or false. If <i>expression</i> evaluates to false, the message: Assertion failed: expr, file ffff, line lnnn is printed to stdout , where ffff is the name of the source file and nnn is the line number where the assertion failed. 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 .	

TYPE	atan (M) - Float Math	<i>ANSI</i>
NAME	atan - return the arc tangent of a double value	
SYNOPSIS	<pre>#include <math.h> double atan (double x);</pre>	
DESCRIPTION	atan() returns the arc tangent of <i>x</i> . <i>x</i> should be specified in radians.	

TYPE	atan2 (M) - Float Math	<i>ANSI</i>
NAME	atan2 - return the arc tangent of a double value y/x	
SYNOPSIS	<pre>#include <math.h> double atan2 (double y, double x);</pre>	
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.	

TYPE	atexit (C) - Miscellaneous	ANSI
NAME	atexit - indicate a function to be called at program termination	
SYNOPSIS	<pre>#include <stdlib.h> int atexit (void (*func) (void));</pre>	
DESCRIPTION	<p>atexit() is used to indicate that the function <i>func</i> should be called at normal program termination. When a program returns from main(), or the exit() function is called, atexit() insures that <i>func</i> is called.</p> <p>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.</p>	
SEE ALSO	exit()	

TYPE	atof (M) - Conversion	<i>ANSI</i>
NAME	atof - convert ASCII string to a double number	
SYNOPSIS	<pre>#include <stdlib.h> double atof(const char *cp);</pre>	
DESCRIPTION	<p>atof() converts a string of text characters pointed at by the argument <i>cp</i> 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.</p>	
EXAMPLE	<pre>#include <stdlib.h> main() { double val; char *cp; cp = "10.6789"; val = atof (cp); printf (" val = %e\n",val); }</pre>	
SEE ALSO	atoi() , atol() , ftoa()	

TYPE	atoi(C) - Conversion	ANSI
NAME	atoi - convert an ASCII string to a signed integer	
SYNOPSIS	<pre>#include <stdlib.h> int atoi (const char *cp);</pre>	
DESCRIPTION	atoi() converts a string of text characters pointed to by the argument <i>cp</i> into a signed integer value, which it returns. The format of the string pointed at by <i>cp</i> should contain three components: optional leading blanks or tabs, followed by an option "+" or "-", followed by an integer. Any numbers following the integer will be ignored, although the string should be null-delimited.	
EXAMPLE	<pre>#include <stdlib.h> main() { int i; char *cp = " 567"; i = atoi (cp); printf ("I = %d\n", i); }</pre>	
SEE ALSO	atof(), atol(), ftoa()	

TYPE	atol (C) - Conversion	<i>ANSI</i>
NAME	atol - convert an ASCII string to a signed long value	
SYNOPSIS	<pre>#include <stdlib.h> long atol (const char *cp);</pre>	
DESCRIPTION	atol() converts the string of characters pointed to by the argument <i>cp</i> 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	<pre>#include <stdio.h> #include <stdlib.h> main() { long val; char *cp = " 79832"; val = atol (cp); printf ("val = %ld\n",val); }</pre>	
SEE ALSO	atof(), atoi(), ftoa()	

TYPE	calloc (C) - Memory Allocation	<i>ANSI</i>
NAME	calloc - allocate space for an array of objects from system memory	
SYNOPSIS	<pre>#include <stdlib.h> void *calloc (size_t nmemb, size_t size);</pre>	
DESCRIPTION	<p>calloc() allocates system memory for an array of <i>nmemb</i> objects, each <i>size</i> bytes long, in a manner similar to the malloc() function. The total size of the block will be <i>size * nmemb</i> bytes long, and each byte within the block is initialized to 0. If calloc() is successful, it returns a pointer to the allocated block.</p>	
SEE ALSO	malloc(), realloc(), free(), lmalloc()	
DIAGNOSTICS	If calloc() cannot allocate a block large enough to hold <i>nmemb</i> elements each <i>size</i> bytes long, it will return a null pointer. Otherwise, a pointer to the requested block is returned.	

TYPE	ceil (M) - Float Math	<i>ANSI</i>
NAME	ceil - compute the smallest integer not less than a specified number	
SYNOPSIS	<pre>#include <math.h> double ceil (double x);</pre>	
DESCRIPTION	ceil() will return the smallest integral number not less than its input, <i>x</i> . The return value is expressed as a double . For example, ceil() will return 6.0 for an input of 5.3, and -5.0 for an input of -5.3.	
SEE ALSO	fabs(), floor()	

TYPE	clearerr (C) - Standard I/O	ANSI
NAME	clearerr - clear end of file and error conditions in a stream	
SYNOPSIS	#include <stdio.h> void clearerr(FILE *stream);	
DESCRIPTION	clearerr() clears both the end-of-file and error condition codes associated with the specified <i>stream</i> . 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()	

TYPE	<code>_cli_parse (C) - Amiga</code>	<i>Amiga</i>
NAME	<code>_cli_parse</code> - parse command line	
SYNOPSIS	<pre>extern int _argc, _arg_len; extern char** _argv, *_arg_lin; void _cli_parse (struct process *pp, long char *aptr)</pre>	
DESCRIPTION	<p><code>_cli_parse()</code> is a function contained within the Aztec library which is called upon starting of a program from the CLI. Its purpose is to parse the command line string passed to the program from the CLI command line and convert it into <code>argc-argv</code> format. <code>_cli_parse()</code> is called by the routine <code>_main</code> with the following arguments:</p> <ul style="list-style-type: none">• <code>pp</code> points to the program's Process structure• <code>aptr</code> points to the command string• <code>alen</code> is the length of the command string. <p><code>_cli_parse</code> parses the command string, allocates memory for the parsed information, and sets up pointers to the parsed information in the global variables <code>_argc</code>, <code>_arg_len</code>, <code>argv</code>, and <code>_arg_line</code>. These variables are used by <code>_main</code> when calling <code>main</code>.</p> <p>This allocated memory is freed by <code>_exit()</code>.</p> <p><code>_cli_parse</code> is supplied as a separate module. Programs that wish to do custom argument parsing may supply their own routine that overrides the one from the library.</p> <p>It is also possible to produce smaller programs by replacing this function with a null function.</p>	

TYPE	clock (C) - Time	<i>ANSI</i>
NAME	clock - determine time intervals	
SYNOPSIS	<pre>#include <time.h> clock_t clock (void);</pre>	
DESCRIPTION	<p>clock() is used to determine the time interval between two events. The 0 value returned by clock() should be divided by the macro CLF_TCK to determine the time in seconds.</p>	

TYPE	close (C) - UNIX I/O	<i>Aztec /UNIX</i>
NAME	close - close a device or file	
SYNOPSIS	#include <fcntl.h> int close (int <i>fd</i>);	
DESCRIPTION	close() closes a device or disk file which has been opened for unbuffered I/O. The parameter <i>fd</i> 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() , <i>fd</i> is the file descriptor returned by open() or creat() . close returns 0 as its value if successful.	
DIAGNOSTICS	If close() fails, it returns -1 and sets an error code in the global integer errno .	

TYPE	cos (M) - Float Math	<i>ANSI</i>
NAME	cos - return the cosine of a double value	
SYNOPSIS	<pre>#include <math.h> double cos (double -x);</pre>	
DESCRIPTION	cos() returns the sine of <i>x</i> . <i>x</i> should be specified in radians.	

TYPE	cosh (M) - Float Math	<i>ANSI</i>
NAME	cosh - return the hyperbolic cosine of a double value	
SYNOPSIS	#include <math.h> double cosh (double <i>x</i>);	
DESCRIPTION	cosh() returns the hyperbolic cosine of <i>x</i> . cosh() will return HUGE_VAL and set errno equal to ERANGE if <i>x</i> is greater than LOGHUGE.	
DIAGNOSTICS	The symbolic values are defined in math.h.	

TYPE	cotan (M) - Float Math										
NAME	cotan - return the cotangent of a double value										
SYNOPSIS	<pre>#include <math.h> double cotan (double x);</pre>										
DESCRIPTION	<p>cotan() returns the cotangent of <i>x</i>. <i>x</i> should be specified in radians. cotan() is not specified by ANSI and may not be portable to other compilers/architectures.</p>										
DIAGNOSTICS	<p>Error Codes:</p> <table><thead><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr></thead><tbody><tr><td> x <TINY_VAL</td><td>-HUGE_VAL (if x<0) HUGE_VAL (if x>0)</td><td>ERANGE</td></tr><tr><td> x >6.74652e⁹</td><td>0.0</td><td>ERANGE</td></tr></tbody></table>		<i>condition</i>	<i>return value</i>	<i>errno</i>	x <TINY_VAL	-HUGE_VAL (if x<0) HUGE_VAL (if x>0)	ERANGE	x >6.74652e ⁹	0.0	ERANGE
<i>condition</i>	<i>return value</i>	<i>errno</i>									
x <TINY_VAL	-HUGE_VAL (if x<0) HUGE_VAL (if x>0)	ERANGE									
x >6.74652e ⁹	0.0	ERANGE									

TYPE	creat (C) - UNIX I/O	<i>Aztec</i>
NAME	creat - create a new file	
SYNOPSIS	#include <fcntl.h> int creat(char *name, int pmode);	
DESCRIPTION	<p>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).</p> <p>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.</p> <p><i>name</i> is a pointer to a character string which is the name of the device or file to be opened.</p> <p>For most systems, <i>pmode</i> 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, <i>pmode</i> should have the octal value 0666.</p> <p>For some systems, <i>pmode</i> is required and has a special meaning.</p>	
DIAGNOSTICS	If creat() fails, it returns -1 as its value and sets a code in the global integer errno .	

TYPE	ctime (C) - Time	ANSI
NAME	ctime - convert a time value to an ASCII string	
SYNOPSIS	<pre>#include <time.h> char *ctime (const time_t *timer);</pre>	
DESCRIPTION	ctime() is equivalent to the asctime() function, except that the time value should be in <code>time_t</code> format rather than <code>tm</code> format. As with asctime(), a pointer to the ASCII time string is returned.	
SEE ALSO	asctime(), time(), localtime()	

TYPE	difftime (C) - Time	ANSI
NAME	difftime - compute the difference between two times	
SYNOPSIS	<pre>#include <time.h> double difftime (time_t time1, time_t time0);</pre>	
DESCRIPTION	difftime() returns the difference of the two time values: <i>time1</i> - <i>time0</i> . The difference is expressed in seconds and is returned as a double.	
SEE ALSO	asctime(), ctime()	

TYPE	div (C) - Integer Math	ANSI
NAME	div - compute the quotient and remainder of the division of two integers	
SYNOPSIS	#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:	
	<pre>typedef struct { int quot; int rem; } div_t;</pre>	
SEE ALSO	ldiv()	

TYPE	dos_packet (C) - Amiga	<i>Amiga</i>
NAME	dos_packet - output packet	
SYNOPSIS	<pre>long dos_packet (port, type, arg1, arg2, arg3, arg4, arg5, arg6, arg7); struct MsgPort *port; long type, arg1, arg2, arg3, arg4, arg5, arg6, arg7;</pre>	
DESCRIPTION	<p>dos_packet() sends a dos packet with the specified type and arguments to the specified port.</p>	

TYPE	exec (C) - Amiga	Amiga
NAME	execl, execv, execlp, execvp - load and start another program	
SYNOPSIS	<pre>execl (char *name, char *arg0, char *arg1, ..., char *argn, 0); execv (char *name, char *argv); execlp (char *name, char *arg0, char *arg1, ..., char *argn, 0); execvp (char *name, char *argv[]);</pre>	
DESCRIPTION	<p>The exec functions can be used within the CLI environment to load and start another program. The called program is loaded on top of the calling program; thus, if the exec function succeeds, it does not return to the caller.</p> <p>Parameters</p> <p><i>name</i> is the name of the file containing the program to be loaded.</p> <p>The exec functions can pass arguments to the called program. execl() and execlp() build a command line by concatenating the strings pointed at by <i>arg1</i>, <i>arg2</i>, and so on. If a C program is being called, its main function will see <i>arg0</i> as <i>argv[0]</i>, <i>arg1</i> as <i>argv[1]</i>, and so on.</p> <p>execv() and execvp() build a command line by concatenating the strings pointed at by <i>argv[0]</i>, <i>argv[1]</i>, and so on. The <i>argv</i> array must have a null pointer as its last entry. If a C program is being called, its main function will see the calling function's <i>argv[i]</i> as its <i>argv[i]</i>.</p>	

The Functions

`exec()` and `execv()` load a program from the specified file. `exec()` is useful when a fixed number of arguments is being passed to a program. `execv()` is useful for programs which are passed a variable number of arguments.

`execl()` and `execvp()` will search for a program first in the current directory, then in the directories specified by the CLI Path command, and then in the C: directory.

Other Features

If an `exec()` function fails, for example because the file does not exist, `exec()` terminates the calling program.

TYPE	exit (C) - Miscellaneous	<i>ANSI</i>
NAME	exit, _exit - terminate calling program	
SYNOPSIS	<pre>#include <stdlib.h> void exit(int code); void _exit(int code);</pre>	
DESCRIPTION	<p>exit() and _exit() terminate the calling program, after releasing all dynamically-allocated memory that was obtained by calling malloc(), calloc(), realloc(), or sbrk(). The functions differ in that exit() closes all files opened for standard and unbuffered I/O, while _exit() does not.</p> <p>If the program was started from within a CLI environment, the program's return code is set to <i>code</i>. Control then returns to a waiting program, which is usually the CLI. When the program was activated by another program's fexec() call, it is this calling program that is waiting and that resumes when the called program exits.</p> <p>If the program was started from the Workbench, then it was executing as an independent process. In this case, the program's process is deleted and the space associated with its stack, segment list, and process structure is released.</p>	

TYPE	exp (M) - Float Math	<i>ANSI</i>
NAME	exp - compute the exponential function e^x	
SYNOPSIS	<pre>#include <math.h> double exp (double x);</pre>	
DESCRIPTION	exp() computes the exponential function of the input parameter x and returns it.	
SEE ALSO	log(), log10(), pow(), sqrt()	
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:	
	<i>condition</i>	<i>return value</i>
	x>LOGHUGE	HUGE_VAL
	x<LOGTINY	0.0
		ERANGE

TYPE	fabs (M) - Float Math	<i>ANSI</i>
NAME	fabs - return the absolute value of a given number	
SYNOPSIS	#include <math.h> double fabs (double <i>x</i>);	
DESCRIPTION	fabs() returns the absolute (or positive) value of the parameter <i>x</i> . Therefore, fabs() will return 5.3 for an input of either 5.3 or -5.3.	
SEE ALSO	floor() , ceil()	

TYPE	fclose (C) - Standard I/O	<i>ANSI</i>
NAME	fclose - close a buffered I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fclose (FILE *stream);</pre>	
DESCRIPTION	<p><code>fclose()</code> causes the specified <i>stream</i> to be flushed and the device or file associated with the <i>stream</i> to be closed. Any data that was written to the <i>stream</i>'s output buffer but not yet written to the file or device will be written by <code>fclose()</code> before closing the file, and the input and output buffers used by the <i>stream</i> will be deallocated. <code>fclose()</code> is automatically called by <code>exit()</code> before exiting the program so that no devices or files are left open after the program terminates.</p>	
SEE ALSO	<code>fopen()</code>	
DIAGNOSTICS	<p><code>fclose()</code> returns 0 if it is successful. If <i>stream</i> is not a valid, open stream, EOF is returned.</p>	

TYPE	<code>fdopen (C) - UNIX I/O</code>	<i>Aztec/UNIX</i>														
NAME	<code>fdopen</code> - open a file or device previously opened for unbuffered I/O for standard I/O															
SYNOPSIS	<pre>#include <stdio.h> FILE *fdopen (int <i>fd</i>, char *<i>mode</i>);</pre>															
DESCRIPTION	<p><code>fdopen()</code> is used to associate a standard I/O stream with a file or device previously opened for unbuffered I/O (with <code>open()</code> or another unbuffered I/O function.) The <i>mode</i> parameter should match the mode with which the file or device was originally opened.</p> <p>The various modes and their meanings are:</p> <table> <thead> <tr> <th><i>Mode</i></th> <th><i>Meaning</i></th> </tr> </thead> <tbody> <tr> <td>a</td> <td>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.</td> </tr> <tr> <td>ab</td> <td>Same as a, except the stream is opened as binary.</td> </tr> <tr> <td>a+</td> <td>Same as a, except the stream may also be read from.</td> </tr> <tr> <td>a+b or ab+</td> <td>Same as a+, except the stream is opened as binary.</td> </tr> <tr> <td>r</td> <td>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.</td> </tr> <tr> <td>rb</td> <td>Same as r, except the stream is opened as binary.</td> </tr> </tbody> </table>	<i>Mode</i>	<i>Meaning</i>	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.	
<i>Mode</i>	<i>Meaning</i>															
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.
SEE ALSO	fopen() , freopen()	
EXAMPLE	<pre>#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) {</pre>	

```
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);
}
```

TYPE	feof (C) - Standard I/O	<i>ANSI</i>
NAME	feof - test for end-of-file in a standard I/O stream	
SYNOPSIS	#include <stdio.h> int feof(FILE *stream);	
DESCRIPTION	feof() is used to test for an end-of-file condition within a specified <i>stream</i> . feof() will return a non-zero value if a <i>stream</i> 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(), fileno(), perror()	

TYPE	ferror (C) - Standard I/O	<i>ANSI</i>
NAME	ferror - test for an error in a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int ferror(FILE *stream);</pre>	
DESCRIPTION	<p>ferror() is used to determine if an I/O error has occurred in the specified <i>stream</i>. ferror() will return a non-zero value if an error has occurred in the <i>stream</i>; 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 <i>stream</i> is closed.</p>	
SEE ALSO	feof() , clearerr() , fileno() , perror()	

TYPE	fexecl, fexecv(C)- Amiga	<i>Aztec</i>
NAME	fexecl,fexecv -load and start another program	
SYNOPSIS	<pre>fexecl (name, arg0,arg1,arg2,...,argn, char*0) char *name, *arg0,*arg2,...,*argn; fexecv (name, argv) char *name, *argv[];</pre>	
DESCRIPTION	<p>The fexec functions load and call another program. The calling program is suspended while the called program is executing; the fexec function returns when the called program terminates.</p> <p><i>wait</i> returns as its value the return code from the fexec-executed program.</p>	

Parameters

name specifies the name of the file containing the program to be loaded, and optionally, the drive on which it's located and the path to it.

The **fexec** functions can pass arguments to the called program. **fexecl()** builds a command line by concatenating the strings pointed at by *arg1*, *arg2*, and so on. If a C program is being called, its main function will see *arg1* as *argv[1]*, *arg2* as *argv[2]*, and so on. *arg0*, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also *argv[0]* of a called C program is always set to a pointer to a null string. Even though *arg0* isn't passed to a program when using Aztec C, it must still be specified, even if it is just a pointer in a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

fexecv() builds a command line by concatenating the strings pointed at by *arcv[1]*, *arcv[2]*, and so on. The *argv* array must have a null pointer as its last entry. If a C program is being called, its main function will see the calling function's *argv[i]* as its *argv[i]. argv[0]*, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also *argv[0]* of a called C program is always a pointer to a null string. Even though *argv[0]* isn't passed to a program when using Aztec C, it must still be specified, even if it is just a pointer to a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

The Functions

fexec() is useful when a fixed number of arguments must be passed, and **fexecv()** is useful when a variable number of arguments must be passed.

Other Information

The **fexec** functions load the called functions after the calling program in memory.

Files opened for unbuffered I/O in the calling program will also be open in the called program, and will have the same file descriptors.

Files opened for standard I/O in the calling program won't be open for standard I/O in the called program, although they will be open for unbuffered I/O. Thus, before a program activates another using an **fexec** function, it should cause the buffered data for files opened for standard I/O to be written to disk, using either the **fclose()** or **fflush()** functions.

The standard input, standard output, and standard error devices are open in the called program to the same devices or files as in the calling program. For the reasons discussed above, care is needed when either the calling or the called program accesses these logical devices using standard I/O calls.

The environment of the called program is the same as that of the calling program.

See Also

The **exec** functions also load programs. Since they overlay the calling program, they allow a larger program to be loaded. They never return to the caller.

Errors

If an **exec** function fails, it returns -1 as its value after setting a code in the global integer *errno*. These codes are defined in the **Errors** Section of the **Library Overview Chapter**.

TYPE	fflush (C) - Standard I/O	<i>ANSI</i>
NAME	fflush - flush an I/O stream	
SYNOPSIS	#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 <i>stream</i> . fflush() is automatically called by fclose() and also any write operation which outputs an end-of-line sequence or causes the <i>stream</i> 's buffer to overflow.	
SEE ALSO	fclose(), fopen()	
DIAGNOSTICS	If fflush() is successful, it returns 0. If a write error occurs, it returns EOF. If <i>stream</i> is NULL, all streams are flushed.	

TYPE	fgetc (C) - Standard I/O	<i>ANSI</i>
NAME	fgetc - return the next available character from a standard I/O stream	
SYNOPSIS	#include <stdio.h> int fgetc(FILE *stream);	
DESCRIPTION	fgetc() returns the next character available from <i>stream</i> and advances the <i>stream</i> '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.	
SEE ALSO	fopen() , fclose() , agetc() , getc() , getchar()	
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.	

TYPE	fgetpos (C) - Standard I/O	ANSI
NAME	fgetpos - save the current file position for a stream	
SYNOPSIS	#include <stdio.h> int fgetpos (FILE *stream, fpos_+ *pos);	
DESCRIPTION	fgetpos() saves the current file position indicator for the specified <i>stream</i> into the object pointed to by <i>pos</i> . The value stored in <i>pos</i> is suitable for use by the fgetpos() function to reposition the <i>stream</i> . 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 <i>errno</i> .	
SEE ALSO	ftell(), fseek(), fsetpos()	

TYPE	fgets (C) - Standard I/O	ANSI
NAME	fgets - get a string of characters from a stream	
SYNOPSIS	<pre>#include <stdio.h> char *fgets (char *s, int n, FILE *stream);</pre>	
DESCRIPTION	<p>fgets() reads in a string of characters from the specified <i>stream</i>. fgets() will place characters from <i>stream</i> into the array pointed to by <i>s</i> until <i>n</i>-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 <i>s</i> is returned by fgets() if no errors occur.</p>	
SEE ALSO	ferror()	
DIAGNOSTICS	If end-of-file is encountered before any characters are read, the array pointed to by <i>s</i> 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.	

TYPE	fileno (C) - UNIX I/O	Aztec
NAME	fileno - return file descriptor associated with a stream	
SYNOPSIS	#include <stdio.h> int fileno(FILE *stream);	
DESCRIPTION	fileno() returns the low-level file descriptor associated with the specified <i>stream</i> . The file descriptor can then be used with the unbuffered I/O functions (<i>open()</i> , <i>read()</i> , etc.)	
SEE ALSO	feof(), ferror(), clearerr(), perror()	

TYPE	floor (M) - Float Math	ANSI
NAME	floor - compute the largest integer not greater than a specified number	
SYNOPSIS	#include <math.h> double floor(double x);	
DESCRIPTION	floor() returns the largest integral value not greater than the input parameter <i>x</i> . 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()	

TYPE	fmod (M) - Float Math	<i>ANSI</i>
NAME	fmod - return the remainder of the double value x/y	
SYNOPSIS	<pre>#include <math.h> double fmod (double x, double y);</pre>	
DESCRIPTION	<p>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 $0 < f < y$.</p>	
SEE ALSO	ceil() , floor() , modf()	

TYPE	fopen (C) - Standard I/O	<i>ANSI</i>
NAME	fopen - open a specified file or device for standard I/O access	
SYNOPSIS	<pre>#include <stdio.h> FILE *fopen (const char *filename, const char *mode);</pre>	
DESCRIPTION	<p>fopen() is used to prepare, or "open", the device or file pointed to by <i>filename</i> for access by the standard I/O functions. Once opened by fopen(), a file or device is referred to as a <i>stream</i>.</p> <p>If the device or file is successfully opened, fopen() returns a pointer, called a <i>file pointer</i>, 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.</p> <p>The first parameter to fopen() is a pointer to the name of the device or file you want to open.</p> <p>The other parameter passed to fopen(), <i>mode</i>, specifies how the file or device is to be accessed. The <i>mode</i> parameter defines two important variables concerning stream accessibility: read/write access and text/binary access.</p> <ul style="list-style-type: none">• 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:

<i>Mode</i>	<i>Meaning</i>
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.

TYPE	format (C) - Conversion	Aztec
NAME	format - write formatted data using a user-defined output function	
SYNOPSIS	<pre>#include <stdio.h> int format (int (*func) (), const char *fmt, va_list varg);</pre>	
DESCRIPTION	<p>format() is used to write formatted ASCII data by calling the function <i>func</i> repeatedly with characters. <i>func</i> should take as an argument a single character, which is a character to be output.</p> <p>The <i>fmt</i> string should have the same format as the function printf(). In fact, printf() could be implemented as:</p> <pre>return (format (putchar, fmt, &args));</pre> <p>See the printf() function for a full description of format()'s conversion process.</p>	
SEE ALSO	va_start(), printf()	

TYPE	fprintf (C, M) - Standard I/O	<i>ANSI</i>
NAME	fprintf - write formatted data to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fprintf (FILE *stream, const char *fmt, ...);</pre>	
DESCRIPTION	<p>fprintf() is used to write formatted ASCII data to the specified <i>stream</i>. The <i>fmt</i> string specifies the exact output to <i>stream</i> and also determines the number of additional arguments that are required. See the printf() function for complete details on the <i>fmt()</i> string.</p>	
SEE ALSO	printf(), format(), sprintf()	

TYPE	fputc (C) - Standard I/O	<i>ANSI</i>
NAME	fputc - write a character to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fputc (int c, FILE *stream);</pre>	
DESCRIPTION	<p>fputc() takes the character <i>c</i> and writes it to the specified I/O stream. Unless an I/O error occurs, fputc() returns <i>c</i>. If an error does occur, fputc() returns EOF. fputc() is identical to the putc() function.</p>	
SEE ALSO	putchar() , putc()	
DIAGNOSTICS	<p>fputc() returns EOF if an error occurs. The actual error code is placed in errno.</p>	

TYPE	fputs (C) - Standard I/O	ANSI
NAME	fputs - write a string to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int fputs (const char *str, FILE *stream);</pre>	
DESCRIPTION	fputs() writes the null-terminated character string pointed to by <i>str</i> to the specified <i>stream</i> . The terminating null in <i>str</i> 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.	

TYPE	fread (C) - Standard I/O	ANSI
NAME	fread - read from a specified standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> size_t fread(void *buffer, size_t size, size_t count, FILE *stream);</pre>	
DESCRIPTION	<p>fread() is used to read one or more characters from <i>stream</i>. fread() will place into the buffer pointed to by <i>buffer</i> up to <i>count</i> items, with each item of size <i>size</i>. The position indicator for <i>stream</i> is advanced by the number of characters that were successfully read. fread() returns as its value the number of items (<i>count</i> if no errors occurred) successfully read from the stream, not the number of characters.</p> <p>See the fwrite() function for an example that uses fread().</p>	
SEE ALSO	fwrite()	
DIAGNOSTICS	<p>fread() returns 0 or a number less than <i>count</i> upon end-of-file or error. The functions feof() and ferror() can be used to distinguish between the two.</p>	

TYPE	free (C) - Memory Allocation	ANSI
NAME	free - deallocate a memory block	
SYNOPSIS	<pre>#include <stdlib.h> void free(void *ptr);</pre>	
DESCRIPTION	<p>free() deallocates a block of memory which was previously reserved with the <code>malloc()</code>, <code>lmalloc()</code>, <code>calloc()</code>, or <code>realloc()</code> functions. This allows the space pointed at by <i>ptr</i> to be used in later attempts to allocate memory. If <i>ptr</i> is a null pointer, free() does nothing. If <i>ptr</i> does not point to a block previously allocated by <code>malloc()</code>, <code>calloc()</code>, or <code>realloc()</code>, or has already been de-allocated by a call to free(), the results are unpredictable.</p>	
SEE ALSO	<code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> , <code>lmalloc()</code>	

TYPE	<code>freeseg (C) - Amiga</code>	<i>Amiga</i>
NAME	<code>freeseg</code> - segment unload function	
SYNOPSIS	<code>void freeseg (int (*func) ());</code>	
DESCRIPTION	<code>freeseg()</code> is used with segmented (overlay) programs to unload a segment that is no longer needed but is occupying memory. The parameter <i>func</i> is the name of any function in the segment. The segment may have been automatically loaded by a function call or explicitly loaded with the <code>segload()</code> function.	
SEE ALSO	<code>segload()</code>	

TYPE	freopen (C) - Standard I/O	ANSI
NAME	freopen - reopen a stream with a new device	
SYNOPSIS	#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 <i>stream</i> with the new file or device <i>filename</i> . freopen() closes the original file or device and returns <i>stream</i> 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() .	
SEE ALSO	fopen()	
EXAMPLE	#include <stdio.h> main() { FILE *fp; fp = freopen ("dskfile", "w+", stdout); printf ("This message is going to dskfile\n"); }	

TYPE	frexp (M) - Float Math	<i>ANSI</i>
NAME	frexp - decompose a floating point number	
SYNOPSIS	#include <math.h> double frexp (double <i>value</i> , int * <i>exp</i>);	
DESCRIPTION	frexp() breaks a floating point number into its component mantissa and exponent. Given <i>value</i> , frexp() places the exponent-portion of <i>value</i> into the buffer pointed to by <i>exp</i> , and returns the mantissa component.	
SEE ALSO	ldexp(), modf()	

TYPE	fscanf (C,M) - Standard I/O	ANSI
NAME	fscanf - perform formatted input conversion on a specified stream	
SYNOPSIS	#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 <i>stream</i> rather than from stdin.	
SEE ALSO	scanf(), sscanf()	

TYPE	fseek (C) - Standard I/O	ANSI
NAME	fseek - reposition current location within a stream	
SYNOPSIS	<pre>#include <stdio.h> int fseek(FILE *stream, long int offset, int origin);</pre>	
DESCRIPTION	<p>fseek() sets the current position within a file opened for standard I/O. <i>stream</i> 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.</p> <p>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 <i>origin</i>. <i>origin</i> may have the following values:</p> <ul style="list-style-type: none">• 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. <p>Offset may be positive or negative.</p> <p>If the file was opened for text mode, the use of fseek() is restricted. Either <i>offset</i> must equal 0, or <i>offset</i> must be a value previously returned by ftell(), with <i>origin</i> set to SEEK_SET.</p> <p>fseek() will clear the end-of-file indicator for the stream and undo the effects of ungetc() calls if successful.</p>	
SEE ALSO	lseek() , ftell()	
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);
}
```

TYPE	fsetpos (C) - Standard I/O	<i>ANSI</i>
NAME	fsetpos - set the correct file position for a stream	
SYNOPSIS	#include <stdio.h> int fsetpos (FILE *stream, const fpos_t *pos);	
DESCRIPTION	fsetpos() sets the file position indicator for the specified <i>stream</i> according to the value contained in the object pointed to by <i>pos</i> . <i>pos</i> should be the value obtained by a previous fgetpos() call. A successful call to fsetpos() will clear the end-of-file indicator for the <i>stream</i> as well as the effects of the unsetc() function on the <i>stream</i> . 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.	
SEE ALSO	ftell(), fseek(), fgetpos()	

TYPE	ftell (C) - Standard I/O	<i>ANSI</i>
NAME	ftell - return the current file position within a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> long ftell(FILE *stream);</pre>	
DESCRIPTION	<p>ftell() returns the current position within the specified <i>stream</i>. 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 <i>stream</i> do not necessarily reflect a measure of characters read or written to the <i>stream</i>.</p>	
SEE ALSO	fseek(), lseek(), fgetpos()	
DIAGNOSTICS	<p>ftell() returns -1L if an error occurs and places the error code in errno. If ftell() is successful, it returns the current file position.</p>	

TYPE	ftoa (M) - Conversion	Aztec
NAME	ftoa - convert floating point number to an ASCII string	
SYNOPSIS	<pre>#include <stdlib.h> void ftoa (double val, char *buf, int precision, int type);</pre>	
DESCRIPTION	<p>ftoa() converts a double-precision floating point number into an ASCII string. The parameter <i>val</i> is the number to be converted, and <i>buf</i> is the buffer where the string is to be placed. It is your responsibility to ensure that the area pointed to by <i>buf</i> is sufficiently large to handle the ASCII representation of <i>val</i>.</p> <p>The <i>precision</i> and <i>type</i> parameters control the format used to convert the number. <i>precision</i> is used to specify the number of digits to be shown to the right of the decimal point. <i>type</i> 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.</p>	
EXAMPLE	<pre>#include <stdio.h> #include <stdlib.h> main() { double val; val = 5.75; ftoa (val, buf, 2, 0); printf ("buf = %s\n", buf); }</pre>	
SEE ALSO	atof(), atoi(), atol()	

TYPE	fwrite (C) - Standard I/O	ANSI
NAME	fwrite - write to a specified standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> size_t fwrite (const void *buf, size_t size, size_t count, FILE *stream);</pre>	
DESCRIPTION	<p>fwrite() is used to write one or more characters to <i>stream</i>. fwrite() takes up to <i>count</i> items, each of size <i>size</i>, from the buffer pointed to by <i>buf</i>. The file position indicator for <i>stream</i> is advanced by the number of characters that were successfully written.</p>	
SEE ALSO	fread()	
DIAGNOSTICS	If a write error occurs, fwrite() will return a number less than <i>count</i> and set an error code in <i>errno</i> .	
EXAMPLE	<p>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:</p> <pre>#include <stdio.h> main() { FILE, *fp1, *fp2; char buf[50]; int size, count, i;</pre>	

```
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, fp
1) != count)
    printf ("Not enough integers
in file1 \n");
fwrite(buf, size, count, fp2);
fclose (fp1);
fclose (fp2);
}
```

TYPE	<code>geta4 (C) - Amiga</code>	<i>Amiga</i>
NAME	<code>geta4</code> - set up a4 register	
SYNOPSIS	<code>void geta4 (void);</code>	
DESCRIPTION	<p><code>geta4()</code> sets up the <code>a4()</code> register that is used as the base addressing register for the small code and data models. It need only be called as the very first item in a newly created task or process.</p> <p>If the code for the <code>geta4()</code> function is physically 32K away from the initial task code, the call to <code>geta4()</code> will not work. In this case, you should either try to place the <code>geta4()</code> routine within 32K of the task code, or compile the task code with the large code option <code>-mc</code>.</p>	

TYPE	getc (C) - Standard I/O	ANSI
NAME	getc - return the next available character from a standard I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int getc(FILE *stream);</pre>	
DESCRIPTION	getc() returns the next character available from <i>stream</i> and advances the <i>stream</i> '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.	
SEE ALSO	fopen(), fclose(), agetc(), fgetc(), getchar()	
DIAGNOSTICS	If an error occurs during the read operation, or if the end-of-file is reached, getc() returns EOF. The functions feof() and perror() may be used to distinguish between a true error and end-of-file.	

TYPE	getchar (C) - Standard I/O	<i>ANSI</i>
NAME	getchar - return the next available character from stdin	
SYNOPSIS	#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).	
SEE ALSO	getc(), fgetc(), fopen(), fclose(), agetc(), putc()	
DIAGNOSTICS	If an error occurs during the read operation, or if the end-of-file is reached, getchar() returns EOF. The functions feof() and perror() may be used to distinguish between the two cases.	

TYPE	getenv (C) - Miscellaneous	ANSI
NAME	getenv - get value of environment variable	
SYNOPSIS	#include <stdlib.h> char *getenv (const char *name);	
DESCRIPTION	<p>getenv() returns a pointer to the character string associated with the environment variable <i>name</i>, or a null pointer if the variable is not in the environment. If the name cannot be found, a null pointer is returned.</p> <p>The character string is in a dynamically-allocated buffer; this buffer will be released when the next call is made to getenv().</p> <p>See the Library Overview chapter for information on environment variables.</p>	
DIAGNOSTICS	If the specified <i>name</i> cannot be found within the environment, a null pointer is returned.	

TYPE	gets (C) - Standard I/O	ANSI
NAME	gets - get a string of characters from stdin	
SYNOPSIS	#include <stdio.h> char *gets (char *buf);	
DESCRIPTION	gets() reads a string of characters from the standard input stream, <code>stdin</code> , into the array pointed to by <code>buf</code> . gets() reads in characters from <code>stdin</code> 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 <code>fgets()</code> function, which preserves the new line. gets() returns <code>buf</code> if no errors occur.	
SEE ALSO	<code>ferror()</code> , <code>fgets()</code> , <code>feof()</code>	
DIAGNOSTICS	If end-of-file is encountered before any characters are read, the array pointed to by <code>buf</code> 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 <code>ferror()</code> and <code>feof()</code> functions may be used to distinguish between an error and the end-of-file.	

TYPE	gmtime (C) - Time	ANSI
NAME	gmtime - convert a calendar time into coordinated universal time	
SYNOPSIS	#include <time.h> struct tm *gmtime (const time_t *timer);	
DESCRIPTION	gmtime() converts a time value in time_t format into struct tm format. The conversion is expressed relative to Greenwich mean time.	
SEE ALSO	localtime(), ctime(), asctime(), time()	

TYPE	index (C) - String Handling	<i>Aztec</i>
NAME	index - find the first occurrence of a character in a string	
SYNOPSIS	#include <string.h> char *index (char *str, char c);	
DESCRIPTION	The index() function returns a pointer to the first occurrence of <i>c</i> within the string pointed to by <i>str</i> . 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()	

TYPE	int_end (C) - Amiga	<i>Amiga</i>
NAME	int_end - restore machine state from within a C interrupt handler	
SYNOPSIS	void int_end(void);	
DESCRIPTION	int_end() is used to restore the machine registers saved by the int_start() function from within a C interrupt handler routine. Calling int_end() is equivalent to the following inline assembly code: <pre>#asm movem.l (Sp)+, d2/d3/d4 #endifasm</pre> int_end() must be the very last line within the C interrupt handling function.	
SEE ALSO	int_start()	

TYPE	int_start (C) - Amiga	<i>Amiga</i>
NAME	int_start - set up a function as an interrupt handler	
SYNOPSIS	void int_start (void);	
DESCRIPTION	<p>int_start() is used in conjunction with the function int_end() to enable a standard C routine to be used as an interrupt handler. Specifically, int_start() saves the d2 and d3 data registers and the a4 address register, as well as calling the geta4() function. This ensures that the C routine does not destroy any register values and also that it can address its global variables properly.</p> <p>The int_start() routine is equivalent to the following in-line assembly code:</p> <pre>#asm movem.1 D2/D3/A4, -(sp) JSR geta4# #endasm</pre> <p>int_start() must be the very first line from within an interrupt routine to guarantee that it will work properly.</p>	
SEE ALSO	int_end()	

TYPE	ioctl (C) - UNIX I/O	<i>Amiga</i>
NAME	ioctl - device I/O utility	
SYNOPSIS	<pre>#include <sgtty.h> ioctl (int <i>fd</i>, int <i>cmd</i>, struct sgttyb *<i>stty</i>);</pre>	
DESCRIPTION	The Amiga version of ioctl() performs only one function: It can set or reset RAW mode on the selected I/O console device. ioctl() is only valid under Version 1.2 or greater of the Amiga operating system.	

TYPE	is... (C) - Character Type	ANSI
NAME	isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii, isgraph, isxdigit - character classification functions	
SYNOPSIS	#include <ctype.h> int isalpha (int c); ...	
DESCRIPTION	<p>These functions test a character value to check if it is of a certain type. If <code>isascii()</code> is true for C, then non-zero (true) will be returned under the following conditions:</p> <p><code>isalpha()</code> c is a letter <code>isupper()</code> c is an uppercase letter <code>islower()</code> c is a lowercase letter <code>isdigit()</code> c is a digit <code>isalnum()</code> c is an alphanumeric character <code>isspace()</code> c is a space, tab, carriage return, newline, form feed or vertical tab <code>ispunct()</code> c is a punctuation character <code>isprint()</code> c is a printing character, valued 0x20 (space) through 0x7e (tilde) <code>iscntrl()</code> c is a delete character (0xff) or ordinary control character (value less than 0x20) <code>isascii()</code> is an ASCII character, code less than 0x100 <code>isgraph()</code> tests for any printing character except a space</p> <p><code>isxdigit()</code> tests for any hexadecimal digit character 0-9, a-f, A-F</p>	

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()`

TYPE	labs (C) - Integer Math	<i>ANSI</i>
NAME	labs - return the absolute value of a signed long	
SYNOPSIS	#include <stdlib.h> long int labs (long int <i>x</i>);	
DESCRIPTION	labs() returns the absolute value of the signed long <i>x</i> .	
SEE ALSO	abs()	

TYPE	lDEXP(M) - Float Math	<i>ANSI</i>
NAME	lDEXP - multiplies a floating point number by an integral power of 2	
SYNOPSIS	<pre>#include <math.h> double lDEXP(double x, int exp)</pre>	
DESCRIPTION	The lDEXP function returns the value of <i>x</i> times 2 raised to the power <i>exp</i> .	

TYPE	ldiv (C) - Integer Math	ANSI
NAME	ldiv - compute the quotient and remainder of the division of two longs	
SYNOPSIS	<pre>#include <stdlib.h> ldiv_t ldiv (long int numerator, long int denominator);</pre>	
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:	
	<pre>typedef struct { long int quot; long int rem; };</pre>	
SEE ALSO	div()	

TYPE	localtime (C) - Time	ANSI
NAME	localtime - convert a time value relative to local time	
SYNOPSIS	<pre>#include <time.h> struct tm *localtime (const time_t *timer);</pre>	
DESCRIPTION	<p>localtime() converts the time value contained in <i>timer</i> and converts it into tm format. The conversion is done relative to the local time.</p>	
SEE ALSO	gmtime(), ctime(), asctime(), time()	

TYPE	log (M) - Float Math	ANSI
NAME	log - compute the natural logarithm of a number	
SYNOPSIS	#include <math.h> double log(double <i>x</i>);	
DESCRIPTION	log() returns as its value the natural logarithm of the input number <i>x</i> .	
SEE ALSO	exp(), log10(), pow(), sqrt()	
DIAGNOSTICS	If the input parameter <i>x</i> 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:	

<i>condition</i>	<i>return value</i>	<i>errno</i>
<i>x == 0</i>	-HUGE_VAL	ERANGE
<i>x < 0</i>	-HUGE_VAL	EDOM

TYPE	log10 (M) - Float Math	<i>ANSI</i>
NAME	log10 - compute the logarithm of a number to base 10	
SYNOPSIS	#include <math.h> double log10 (double <i>x</i>);	
DESCRIPTION	log10 returns the logarithm to base 10 of the input parameter <i>x</i> .	
SEE ALSO	exp(), log(), pow(), sqrt()	
DIAGNOSTICS	If the input parameter <i>x</i> is negative, log10 will return -HUGE_VAL and set errno equal to ENOM. If the input value <i>x</i> is 0, -HUGE_VAL is returned, and errno is set to ERANGE.	
	Error codes:	
	<i>condition</i>	<i>return value</i>
	<i>x == 0</i>	-HUGE_VAL
	<i>x < 0</i>	-HUGE_VAL
		<i>errno</i>
		ERANGE
		EDOM

TYPE	longjmp (C) - Miscellaneous	ANSI
NAME	longjmp - execute a non-local goto	
SYNOPSIS	#include <setjmp.h> void longjmp (jmp_buf <i>env</i> , int <i>retval</i>);	
DESCRIPTION	A call to the longjmp() function restores the stack and register state saved in the last call to setjmp() with the argument <i>env</i> , and then executes a return which makes it appear that setjmp() returned the value <i>retval</i> . It is crucial that setjmp() be called with <i>env</i> before any longjmp() calls occur. If setjmp() is not called with <i>env</i> 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 <i>retval</i> . If <i>retval</i> 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()	

TYPE	lseek (C) - UNIX I/O	<i>Aztec /UNIX</i>
NAME	lseek - change current position within file	
SYNOPSIS	long int lseek (int <i>fd</i>, long <i>offset</i>, int <i>origin</i>);	
DESCRIPTION	<p>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.</p> <p><i>fd</i> 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:</p> <ul style="list-style-type: none">• If <i>origin</i> is 0, the current position is set to <i>offset</i> bytes from the beginning of the file.• If <i>origin</i> is 1, the current position is set to the current position plus <i>offset</i>.• If <i>origin</i> is 2, the current position is set to the end of the file plus <i>offset</i>. <p>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.</p> <p>If lseek() is successful, it returns the new position in the file (in bytes from the beginning of the file).</p>	
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);
```

TYPE	malloc (C) - Memory Allocation	<i>ANSI</i>
NAME	malloc - allocate a block of system memory	
SYNOPSIS	<pre>#include <stdlib.h> void *malloc (size_t size);</pre>	
DESCRIPTION	<p>malloc() allocates a block of memory <i>size</i> 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.</p> <p>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.</p> <p>If the allocation is successful, malloc() returns a pointer to the requested block.</p>	
SEE ALSO	calloc(), realloc(), free(), lmalloc()	
DIAGNOSTICS	If malloc() cannot allocate the requested <i>size</i> block, it returns a null pointer. Otherwise, a pointer to the requested block is returned.	

TYPE	memchr (C) - Block Operation	<i>ANSI</i>
NAME	memchr - find a character within an object	
SYNOPSIS	#include <string.h> void *memchr (const void *obj, int c, size_t n);	
DESCRIPTION	memchr() searches the first <i>n</i> bytes in the object pointed to by <i>obj</i> for the character <i>c</i> . If <i>c</i> is found, memchr() returns a pointer to it. Otherwise, a NULL pointer is returned.	
SEE ALSO	strrchr(), strchr()	

TYPE	memcmp (C) - Block Operation	ANSI
NAME	memcmp - compare two blocks of memory <i>n</i> bytes long	
SYNOPSIS	<pre>#include <string.h> int memcmp (const void *blk1, const void *blk2, size_t n);</pre>	
DESCRIPTION	The <code>memcmp()</code> function compares the first <i>n</i> characters in the object pointed to by <i>blk1</i> with the first <i>n</i> characters in the object pointed to by <i>blk2</i> . <code>memcmp()</code> returns a positive number, negative number, or zero, depending on whether <i>blk1</i> is greater than, less than, or equal to <i>blk2</i> .	
SEE ALSO	<code>strcmp()</code> , <code>strncmp()</code> , <code>strcoll()</code>	

TYPE	memcpy (C) - Block Operation	<i>ANSI</i>
NAME	memcpy - copy a block of bytes from one object to another	
SYNOPSIS	<pre>#include <string.h> void *memcpy void *dest, const void *src, size_t n);</pre>	
DESCRIPTION	memcpy() copies the first <i>n</i> bytes from the object pointed to by <i>src</i> into the object pointed to by <i>dest</i> . The two objects should <i>not</i> overlap. memcpy() returns <i>dest</i> .	
SEE ALSO	memmove(), strcpy(), strncpy()	

TYPE	memmove (C) Block Operation	ANSI
NAME	memmove - copy a block of memory	
SYNOPSIS	<pre>#include <string.h> void *memmove (void *dest, const void *source, size_t n);</pre>	
DESCRIPTION	<p>memmove() copies a block of data from one location in memory to another location. memmove() will copy <i>n</i> characters from the object pointed to by <i>source</i> to the object pointed to by <i>dest</i>. memmove() acts as if the data pointed to by <i>source</i> is first copied into a temporary buffer before moving it into <i>dest</i>. Therefore, overlapping blocks may be used with this function. memmove() returns the value of <i>dest</i>.</p>	
SEE ALSO	memcpy()	

TYPE	memset (C) - Block Operation	<i>ANSI</i>
NAME	memset - set a block of memory to a specified value	
SYNOPSIS	#include <string.h> void memset (void *obj, int c, size_t n);	
DESCRIPTION	memset() copies the value of <i>c</i> into the first <i>n</i> bytes of the object pointed to by <i>obj</i> .	
SEE ALSO	memcpy(), memcmp()	

TYPE	mktemp (C) - UNIX I/O	<i>Amiga</i>
NAME	mktemp - make a unique file name	
SYNOPSIS	char *mktemp (char *template);	
DESCRIPTION	mktemp() replaces the character string pointed at by <i>template</i> with the name of a nonexistent file and returns as its value a pointer to the string.	The string pointed at by <i>template</i> should look like a filename whose last few characters are Xs with an optional imbedded period. mktemp() replaces the Xs with a letter followed by the least significant digits of the starting address of its program's data segment. The letter will be between A and Z and will be chosen such that the resulting character string is not the name of an existing file.
DIAGNOSTICS	For a given character string, mktemp() will try to convert the string into one of 26 file names. If all of these files exist, mktemp() will replace the first character pointed at by <i>template</i> with a null character.	
SEE ALSO	tmpfile(), tmpnam()	
EXAMPLES	The following program calls mktemp() to get a character string that it can use as a filename. If the digits selected by mktemp() are 1234, then the generated name will be one of the strings abcA001.234 , abcB001.234 , .. abcZ001.234 . If all the strings that mktemp() considers are names of existing files, mktemp() will replace the first character of the string passed to it, a in this case, with 0.	

```
#include <stdio.h>
main()
{
    char *fname, *mktemp();
    FILE *fp, fopen();
    fname=mktemp("abcXXX.XXX");
    if (!*fname){
        printf("mktemp failed");
        exit(1);
    } else
        fp=fopen(fname, "w");
    ...
}
```

TYPE	mktme (C) - Time	<i>ANSI</i>
NAME	mktme - convert a time value between formats	
SYNOPSIS	<pre>#includes <time.h> time_t mktme (struct tm *timeptr);</pre>	
DESCRIPTION	<p>mktme() translates a time value represented in struct tm format into time_t format. <i>timeptr</i> should be a pointer to the tm format time value, and the corresponding time_t value is returned by mktme(). If <i>timeptr</i> is unconvertible, mktme() returns (time_t) -1.</p>	
SEE ALSO	asctime(), ctime(), localtime(), time()	

TYPE	modf (M) - Float Math	<i>ANSI</i>
NAME	modf - break a floating point value into integral and fractional parts	
SYNOPSIS	#include <math.h> double modf(double <i>val</i> , double * <i>iptr</i>);	
DESCRIPTION	modf() breaks the floating point number <i>val</i> into its component 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 <i>iptr</i> , and the fractional portion is returned as the return value of modf().	
SEE ALSO	frexp(), ldexp()	

TYPE	movmem(C) Block Operation	<i>Aztec</i>
NAME	movmem - copy a memory block	
SYNOPSIS	<pre>void movmem(source, dest, length) void *source, *dest; unsigned int length;</pre>	
DESCRIPTION	<p>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.</p>	
SEE ALSO	memcpy, memmove	
APPLICATION NOTE	<p>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.</p>	

TYPE	open (C) - UNIX I/O	Aztec /UNIX															
NAME	open - open device or file for unbuffered I/O																
SYNOPSIS	#include <fcntl.h> open (char *name, int mode);																
DESCRIPTION	<p>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.</p> <p><i>name</i> is a pointer to a character string which is the name of the device or file to be opened.</p> <p><i>mode</i> specifies how your program intends to access the file. The choices are as follows:</p> <table><thead><tr><th>Mode</th><th>Meaning</th></tr></thead><tbody><tr><td>O_RDONLY</td><td>read only</td></tr><tr><td>O_WRONLY</td><td>write only</td></tr><tr><td>O_RDWR</td><td>read and write</td></tr><tr><td>O_CREAT</td><td>create file, then open it</td></tr><tr><td>O_TRUNC</td><td>truncate file, then open it</td></tr><tr><td>O_EXCL</td><td>cause open() to fail if file already exists; used with O_CREAT</td></tr><tr><td>O_APPEND</td><td>position file for appending data</td></tr></tbody></table> <p>These open modes are integer constants defined in the files 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.</p> <p>The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, and O_RDWR in the <i>mode</i> parameter. The three remaining values</p>	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
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																

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 the end of the file.

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.

DIAGNOSTICS If **open** encounters an error, it returns -1 and sets the global integer **errno** to a symbolic value which identifies the error.

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);
}
```

TYPE	perror (C) - Standard I/O	ANSI																																	
NAME	perror - print a system error message																																		
SYNOPSIS	<pre>#include <stdio.h> void perror(const char *str);</pre>																																		
DESCRIPTION	<p>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 <code>errno</code>. The <code>perror()</code> function is used to print an error message to the <code>stderr()</code> stream which corresponds to the error indicated by <code>errno</code>. The exact format of the message written to <code>stderr()</code> is:</p> <ul style="list-style-type: none">• The string pointed at by <code>str</code> (only if <code>str</code> 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. <p>On the Amiga, the possible values of <code>errno</code> and their corresponding error messages are:</p> <table><thead><tr><th><i>Errno</i></th><th><i>Value</i></th><th><i>message</i></th></tr></thead><tbody><tr><td>0</td><td></td><td>Error 0</td></tr><tr><td>ENOENT</td><td></td><td>No such file or directory</td></tr><tr><td>E2BIG</td><td></td><td>Arg list too long</td></tr><tr><td>EBADF</td><td></td><td>Bad file descriptor</td></tr><tr><td>ENOMEM</td><td></td><td>Not enough memory</td></tr><tr><td>EEXIST</td><td></td><td>File exists</td></tr><tr><td>EINVAL</td><td></td><td>Invalid argument</td></tr><tr><td>ENFILE</td><td></td><td>File table overflow</td></tr><tr><td>EMFILE</td><td></td><td>Too many open files</td></tr><tr><td>ENOTTY</td><td></td><td>Not a console</td></tr></tbody></table>	<i>Errno</i>	<i>Value</i>	<i>message</i>	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	LIBRARY FUNCTIONS
<i>Errno</i>	<i>Value</i>	<i>message</i>																																	
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																																	

EACCES	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

TYPE	pow (M) - Float Math	<i>ANSI</i>
NAME	pow - compute x to the y <i>th</i> power	
SYNOPSIS	#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 <i>th</i> power (x^y).	
SEE ALSO	exp() , log() , log10() , sqr()	
DIAGNOSTICS	If $x = 0$ and $y \leq 0$, 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.	

TYPE	printf (C, M) - Standard I/O	ANSI
NAME	printf - formatted output function	
SYNOPSIS	<pre>#include <stdio.h> int printf (const char *fmt, ...);</pre>	
DESCRIPTION	<p>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 <code>stdout</code> stream.</p>	

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 following 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 or 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.
- .
- 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

- d** **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

<i>Character</i>	<i>Argument Type</i>	<i>Conversion</i>
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 (that is, the address of the object is printing at) 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:

```
ln ... -ln -lc
```

TYPE	putc (C) - Standard I/O	ANSI
NAME	putc - write a character to an I/O stream	
SYNOPSIS	<pre>#include <stdio.h> int putc (int c, FILE *stream);</pre>	
DESCRIPTION	putc() takes the character <i>c</i> and writes it to the specified I/O stream. Unless an I/O error occurs, putc() returns <i>c</i> . If an error does occur, putc() returns EOF. This function is identical to fputc().	
SEE ALSO	putchar(), fputc()	
DIAGNOSTICS	putc() returns EOF if an error occurs. The actual error code is placed in errno.	

TYPE	putchar (C) - Standard I/O	ANSI
NAME	putchar - write a character to the <code>stdout</code> stream	
SYNOPSIS	<pre>#include <stdio.h> int putchar (int c);</pre>	
DESCRIPTION	<p><code>putchar()</code> is identical to the function <code>putc()</code>, except that it always writes to <code>stdout</code>, i.e., it is the same as</p> <pre>putc (stdout)</pre>	
DIAGNOSTICS	<p><code>putchar()</code> returns EOF if an I/O occurred during the write. The error code is set in the global integer <code>errno</code>.</p>	
SEE ALSO	<code>putc()</code> , <code>fputc()</code>	

TYPE	puts (C) - Standard I/O	<i>ANSI</i>
NAME	puts - write a string to stdout	
SYNOPSIS	<pre>#include <stdio.h> int puts(const char *str);</pre>	
DESCRIPTION	<p>puts writes the null-terminated character string pointed to by <i>str</i> 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.</p>	
DIAGNOSTICS	<p>EOF is returned by puts() if an error occurs, with the error number contained in errno.</p>	

TYPE	qsort (C) - Miscellaneous	Aztec
NAME	qsort - sort an array of records in memory	
SYNOPSIS	<pre>#include <stdlib.h> void qsort (void *array, size_t num, size_t width, int (*func)());</pre>	
DESCRIPTION	<p>qsort() sorts an array of elements using Hoare's Quicksort algorithm. <i>array</i> is a pointer to the array to be sorted; <i>num</i> is the number of records to be sorted; <i>width</i> is the size in bytes of each array element; <i>func</i> is a pointer to a function that is called for a comparison of two array elements. This function must be written by you.</p> <p><i>func</i> is passed pointers to the two elements being compared. It must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.</p>	
EXAMPLE	The Aztec linker, <i>In</i> , 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);
}
```

TYPE	ran (M) - Float Math	<i>Aztec</i>
NAME	ran - generate floating point random numbers	
SYNOPSIS	#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()	

TYPE	rand (C) - Integer Math	<i>ANSI</i>
NAME	rand - return a pseudo-random integer	
SYNOPSIS	#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()	

TYPE	read (C) - UNIX I/O	Aztec /UNIX
NAME	read - read from a device or file using unbuffered I/O	
SYNOPSIS	<pre>#include <fcntl.h> int read (int fd, void *buf, size_t bufsize);</pre>	
DESCRIPTION	<p>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.</p> <p><i>fd</i> is the file descriptor which was returned to the caller when the device or file was opened.</p> <p><i>buf</i> is a pointer to the buffer into which the information is to be placed.</p> <p><i>bufsize</i> is the number of characters to be transferred.</p> <p>If read() is successful, it returns as its value the number of characters transferred.</p> <p>If the returned value is zero, then end-of-file has been reached, immediately, with no bytes read.</p>	
SEE ALSO	open(), close()	
DIAGNOSTICS	If the operation is not successful, read() returns -1 and places a code in the global integer errno .	

TYPE	realloc (C) - Memory Allocation	ANSI
NAME	realloc - re-allocate an existing memory block to a different size	
SYNOPSIS	<pre>#include <stdlib.h> void *realloc (void *ptr, size_t size);</pre>	
DESCRIPTION	<p>realloc() changes the size of a memory block, <i>ptr</i>, 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.</p> <p>If <i>size</i> 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 <i>size</i> is 0, realloc() deallocates the block in a manner similar to the free() function. If <i>ptr</i> is 0, realloc() behaves like the malloc() function and allocates a new block of length <i>size</i>.</p> <p>If <i>ptr</i> does not point to a block previously allocated by malloc(), calloc(), or realloc(), or if <i>ptr</i> was deallocated by the free() function, the behavior of realloc() is unpredictable.</p>	
SEE ALSO	malloc() , calloc() , free() , lmalloc()	
DIAGNOSTICS	If realloc() cannot find a block at least <i>size</i> bytes in length available, it returns a null pointer. Otherwise, a pointer to the requested block is returned.	

TYPE	remove (C) - Standard I/O	<i>ANSI</i>
NAME	remove - delete a file	
SYNOPSIS	#include <stdio.h> int remove (const char *filename);	
DESCRIPTION	remove() deletes the disk file named <i>filename</i> from the disk. remove() returns 0 if it is successful, and non-zero if it is not.	
SEE ALSO	unlink()	
DIAGNOSTICS	If an error occurs, remove() sets <i>errno</i> with the error code and returns a non-zero value.	

TYPE	rename (C) - Standard I/O	ANSI
NAME	rename - rename a disk file	
SYNOPSIS	#include <stdio.h> int rename (const char *old, const char *new);	
DESCRIPTION	<p>rename() changes the name of a file. <i>old</i> is a pointer to a null-terminated string containing the old file name, and <i>new</i> is a pointer to a null-terminated string containing the new name of the file.</p> <p>If successful, rename() returns 0 as its value; if unsuccessful, it returns EOF.</p> <p>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.</p>	

TYPE	rewind (C) - Standard I/O	ANSI
NAME	rewind - reposition a stream's position indicator to the beginning of the file	
SYNOPSIS	<pre>#include <stdio.h> void rewind (FILE *stream);</pre>	
DESCRIPTION	rewind() sets the indicated <i>stream</i> 's file position indicator to the beginning of the file. rewind() is equivalent to the call: <code>(void) fseek (stream, 0L, SEEK_SET);</code> with the exception that rewind() also clears the error indicator for the <i>stream</i> .	
SEE ALSO	fseek()	

TYPE	rindex (C) - String Handling	<i>Aztec</i>
NAME	rindex - find the last occurrence of a character in a string	
SYNOPSIS	#include <string.h> char *rindex (char *str, char c);	
DESCRIPTION	<p>rindex() returns a pointer to the last occurrence of <i>c</i> within the string pointed to by <i>str</i>. If the character is not found in <i>str</i>, then NULL is returned.</p> <p>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.</p>	
SEE ALSO	strchr(), strrchr(), index()	

TYPE	sbrk (C) - Memory Allocation	Aztec
NAME	sbrk - memory allocation function	
SYNOPSIS	#include <stdlib.h> void *sbrk (size_t size);	
DESCRIPTION	<p>sbrk(), which is contained along with alternate versions of other memory-allocation functions in the file heapmem.o, provides an elementary means of allocating and deallocating space from a contiguous block of memory. When first called, sbrk() gets a block of memory by calling the Amiga function AllocMem. The default size of this block is 40K bytes. A program can specify a different size by setting the desired size in the global long _Heapsize before issuing its first call to a memory-allocation function or to a standard I/O function.</p> <p>sbrk() maintains a pointer to the top of allocated space within the block. When called, sbrk() increments this pointer by <i>size</i> bytes and returns the value that the pointer had on entry.</p> <p>When a program terminates, the block of memory that was allocated by sbrk(), if any, is automatically released.</p>	
ERRORS	If an sbrk() request would make the sbrk() pointer go past the end of sbrk() 's block of memory, sbrk() will return -1 as its value, without modifying its pointer.	

TYPE	scanf (C,M) - Standard I/O	ANSI
NAME	scanf - perform formatted input conversion on the <code>stdin</code> stream	
SYNOPSIS	<pre>#include <stdio.h> int scanf (const char *fmt, ...);</pre>	
DESCRIPTION	<p><code>scanf()</code> converts a <i>stream</i> of text characters from the <i>stream</i> as directed by the control string pointed to by the <i>fmt</i> parameter and places the results in the additional pointer parameters. There must be the same number of format specifiers in the <i>fmt</i> string as pointer arguments.</p>	

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 pointer to int.
- u unassigned decimal integer. The argument corresponding to this item should be pointer 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 pointer 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**.

SEE ALSO

fscanf(), **sscanf()**

TYPE	scdir (C) - Amiga	<i>Amiga</i>
NAME	scdir - return the name of the next file matching pattern	
SYNOPSIS	#include <fcntl.h> char * scdir (char *pat);	
DESCRIPTION	<p>scdir() is a function that permits you to perform wildcard expansion on filename patterns using native Amiga facilities.</p> <p>When scdir() is called with a pattern, it returns a pointer to a static area containing the null terminated name of the next file that matches the pattern, or zero if no more files match the pattern. Since the area containing the name is statically allocated, the name will be overwritten by subsequent calls to scdir().</p> <p>Patterns have two wildcard characters. The asterisk (*) matches any number of characters. The question mark (?) matches a single character. Note that scdir() uses UNIX wild-card conventions, not AmigaDos conventions.</p>	
EXAMPLE	<pre>main() { char *sav[100]; register char *pat; register int i; /* get all .c files on current dir */ pat = "*.c"; i = 0; while ((ptr = scdir(pat)) && i < 100) { sav[i] = malloc (strlen(ptr)+1); strcpy(sav[i++], ptr); /* rest of program */ }</pre>	

TYPE	screen (S) - Amiga	<i>Amiga</i>
NAME	scr_beep, scr_bs, scr_tab, scr_lf, scr_cursup, scr_cursrt, scr_cr, scr_clear, scr_home, scr_curs, scr_eol, scr_linsert, scr_ldelete, scr_cinsert, scr_cdelete - screen manipulation functions	
SYNOPSIS	void scr_beep (void) void scr_bs (void) void scr_tab (void) void scr_lf (void) void scr_cursup (void) void scr_cursrt (void) void scr_cr (void) void scr_clear (void) void scr_home (void) void scr_eol (void) void scr_linsert (void) void scr_ldelete (void) void scr_cinsert (void) void scr_cdelete (void) void scr_curs (<i>lin, col</i>) void int <i>lin, col</i> ;	
DESCRIPTION	<p>These functions can be called by command programs to manipulate screens of text. For example, there are functions to clear the screen, position the cursor, and insert and delete characters and lines.</p> <p>These functions can be used in conjunction with the normal standard I/O and unbuffered I/O functions to display characters on the console.</p>	

scr_beep() rings the keyboard bell.

scr_bs() moves the cursor back one character space without modifying the character that was backspaced over.

scr_tab() moves the cursor right one tab stop.

scr_lf() moves the cursor down one line, scrolling if at the bottom of the screen.

scr_cursup() moves the cursor up without changing its column location.

scr_cursrt() moves the cursor right one character space, without modifying the character that was spaced over.

scr_cr() causes a carriage return.

scr_clear() clears the screen and homes the cursor.

scr_home() homes the cursor to the upper left hand corner of the screen.

scr_curs() moves the cursor to the line and column specified by the *lin* and *col* parameters, respectively.

scr_eol() erases the line at which the cursor is located, from the current cursor position to the end of the line.

scr_linsert() inserts a blank line at the cursor location, moving the lines below the cursor down one line.

scr_ldelete() deletes the line at the cursor location, moving the lines below the cursor up one line and placing a blank line at the bottom of the screen.

scr_cinsert() inserts a space at the cursor location, shifting right one character the characters in the line which are on the right of the cursor.

scr_cdelete() deletes the character at the cursor location, shifting left one character the characters in the line which are on the right of the cursor.

TYPE	segload (C) - Amiga	<i>Amiga</i>
NAME	segload - load a program segment	
SYNOPSIS	<code>void segload (int (*func)());</code>	
DESCRIPTION	<p><code>segload()</code> is used with segmented (overlay) programs to force an overlay segment into memory without actually calling any of the functions within it. Normally, when a function is called that is not in memory, its segment is automatically loaded and the function is then jumped to.</p> <p>The argument <i>func</i> should be the name of a function within the segment you want to load. The <code>segload()</code> function is often used by a program when it detects a sufficient amount of free memory is available to hold the entire program. <code>segload()</code> is called to load all of the program's segments at the start of the program, which puts all of the load delay just at the beginning.</p>	
SEE ALSO	<code>freeseg()</code>	

TYPE	setbuf (C) - Standard I/O	ANSI
NAME	setbuf - associate an I/O stream with a specific buffer	
SYNOPSIS	#include <stdio.h> void setbuf (FILE *stream, char *buf);	
DESCRIPTION	setbuf() is equivalent to setvbuf() function called in the following manner: <ul style="list-style-type: none">• If <i>buf</i> is not null, then setbuf() is the same as setvbuf() called with <i>_IOFBF</i> (fully buffered) for <i>mode</i> and <i>BUFSIZE</i> (defined in <i>stdio.h</i>) for <i>size</i>. Note that this means that your buffer <i>must</i> be at least <i>BUFSIZE</i> in length.• If <i>buf</i> is null, then setbuf() is the same as setvbuf() called with <i>_IONBF</i> (no buffering) for <i>mode</i>.	
SEE ALSO	setvbuf()	

TYPE	setenv (C) - Miscellaneous	<i>Amiga</i>
NAME	setenv - set environment variables	
SYNOPSIS	void setenv (char *name, char *buf);	
DESCRIPTION	This function adds or deletes environment variables to or from the pseudo-library used to implement the environment. <i>name</i> is a pointer to the environment variable to be changed and <i>buf</i> is a pointer to the new value. If <i>buf</i> points to a null string, the variable <i>name</i> will be removed from the environment. If the environment does not already exist, it will be created.	

TYPE	setjmp - Miscellaneous	ANSI
NAME	setjmp(C) - set up for a non-local goto	
SYNOPSIS	#include <setjmp.h> int setjmp (jmp_buf env);	
DESCRIPTION	<p>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.</p> <p>The argument <i>env</i> should be declared as an instance of the type <i>jmp_buf</i>. setjmp() saves the current stack and register variable information in <i>env</i> so that this information may be restored upon invocation of a longjmp(), and returns a 0.</p>	
SEE ALSO	longjmp()	

TYPE	setvbuf (C) - Standard I/O	ANSI						
NAME	setvbuf - associate an I/O stream with a specific buffer							
SYNOPSIS	#include <stdio.h> int setvbuf(FILE *stream, char *buf, int mode, size_t size);							
DESCRIPTION	<p>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.</p> <p>The type of buffering to be performed on the stream is determined by the <i>mode</i> argument:</p> <table><tr><td>_IOFBF</td><td>The stream is <i>fully buffered</i>. 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.</td></tr><tr><td>_IOLBF</td><td>The stream is <i>line buffered</i>. 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.</td></tr><tr><td>_IONBF</td><td>The stream is <i>unbuffered</i>, and the <i>buf</i> and <i>size</i> 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.</td></tr></table> <p>You must insure that <i>buf</i> stays in existence throughout the time the stream is open. This means that if <i>buf</i> is a local array, the stream must be closed before the function returns. Also, you</p>	_IOFBF	The stream is <i>fully buffered</i> . 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 <i>line buffered</i> . 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 <i>unbuffered</i> , and the <i>buf</i> and <i>size</i> 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.	
_IOFBF	The stream is <i>fully buffered</i> . 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 <i>line buffered</i> . 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 <i>unbuffered</i> , and the <i>buf</i> and <i>size</i> 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.							

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()`

TYPE	signal (C) - Signal	<i>ANSI</i>
NAME	signal - define how to handle a signal	
SYNOPSIS	#include <signal.h> void (*signal (int <i>sig</i> , void (* <i>func</i>) (int)) (int);	
DESCRIPTION	signal() specifies that the signal whose number is <i>sig</i> is to be handled as defined by function. 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.

When a program activates another program, by calling one of the `exec` or system functions, the *func* for the signals of the called program are initially set to SIG_DFL regardless of their setting in the calling program. If the called program returns to the calling program, the *func* for the signals in the calling program resume the value that they had just prior to the activation of the called program.

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);
}
```

TYPE	sin (M) - Float Math	ANSI						
NAME	sin - return the sine of a double value							
SYNOPSIS	#include <math.h> double sin (double x);							
DESCRIPTION	sin() returns the sine of x . x should be specified in radians. Error codes: <table><thead><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr></thead><tbody><tr><td>$x > 6.7465e^9$</td><td>0.0</td><td>ERANGE</td></tr></tbody></table>	<i>condition</i>	<i>return value</i>	<i>errno</i>	$ x > 6.7465e^9$	0.0	ERANGE	
<i>condition</i>	<i>return value</i>	<i>errno</i>						
$ x > 6.7465e^9$	0.0	ERANGE						

TYPE	sinh (M) - Float Math	ANSI						
NAME	sinh - return the hyperbolic sine of a double value							
SYNOPSIS	#include <math.h> double sinh(double <i>x</i>);							
DESCRIPTION	sinh() returns the hyperbolic sine of <i>x</i> . sinh() will return HUGE_VAL and set errno equal to ERANGE if <i>x</i> is greater than 348.6. Error codes: <table><thead><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr></thead><tbody><tr><td> <i>x</i> > LOGHUGE+ ~0.6932</td><td>HUGE_VAL</td><td>ERANGE</td></tr></tbody></table>	<i>condition</i>	<i>return value</i>	<i>errno</i>	<i>x</i> > LOGHUGE+ ~0.6932	HUGE_VAL	ERANGE	
<i>condition</i>	<i>return value</i>	<i>errno</i>						
<i>x</i> > LOGHUGE+ ~0.6932	HUGE_VAL	ERANGE						

TYPE	sprintf (C, M) - Conversion	ANSI
NAME	sprintf - write formatted data to a buffer	
SYNOPSIS	#include <stdio.h> int sprintf (char *buf, const char *fmt, ...);	
DESCRIPTION	sprintf() is used to write formatted ASCII data to the specified buffer <i>buf</i> . The <i>fmt</i> string specifies the exact contents of <i>buf</i> and also determines the number of additional required arguments. See the printf() function for complete details on the <i>fmt</i> string.	
SEE ALSO	printf(), fprintf(), format()	

TYPE	sqrt (M) - Float Math	<i>ANSI</i>
NAME	sqrt - compute the non-negative square root of a value	
SYNOPSIS	#include <math.h> double sqrt(double x);	
DESCRIPTION	sqrt() returns the nonnegative square root of the input parameter <i>x</i> .	
SEE ALSO	exp(), log(), log10(), pow()	
DIAGNOSTICS	If <i>x</i> is negative, errno is set to EDOM and sqrt() returns 0.0 as its value.	

TYPE	sran (M) - Float Math	<i>Aztec</i>
NAME	sran - set the random number seed for ran	
SYNOPSIS	<pre>#include <math.h> void sran (double seed);</pre>	
DESCRIPTION	<p>sran() is used to set the seed-value for the function ran(). sran() is not defined by ANSI.</p>	
SEE ALSO	ran(), rand(), srand()	

TYPE	srand (C) - Integer Math	<i>ANSI</i>
NAME	srand - initialize the seed value used by rand	
SYNOPSIS	<pre>#include <stdlib.h> void srand (unsigned int seed);</pre>	
DESCRIPTION	The value <i>seed</i> 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 <i>seed</i> , 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 <i>seed</i> of 1.	
SEE ALSO	rand() , ran() , sran()	

TYPE	sscanf (C,M) - Conversion	<i>ANSI</i>
NAME	sscanf - perform formatted input conversion on a buffer	
SYNOPSIS	<pre>#include <stdio.h> int sscanf (const char *str, const char *fmt, ...);</pre>	
DESCRIPTION	sscanf() is identical to the scanf() function, except that sscanf() reads from the buffer pointed to by <i>str</i> rather than from stdin .	
SEE ALSO	scanf(), fscanf()	

TYPE	stat (C) - UNIX I/O	<i>Amiga</i>
NAME	stat - return file information	
SYNOPSIS	<pre>#include <stat.h> #include <time.h> struct stat *stat(char *name, struct stat *buf);</pre>	
DESCRIPTION	<p>stat() returns the attribute byte, date and time, and size of the <i>filename</i>. This information is returned in <i>buf</i>, which has the following format:</p> <pre>struct stat { char st_attr; long st_mtime; long st_size; long st_rsize; };</pre> <p>This structure, and the meaning of the bits in the attribute and time fields, are defined in the header files stat.h and time.h.</p> <p><i>name</i> can optionally specify the full pathname where the file is located.</p>	
ERRORS	stat() returns -1 if it fails, after setting a code in the global integer errno .	

TYPE	_stkchk (C) - Miscellaneous	<i>Amiga</i>
NAME	_stkchk - check for stack overflow	
SYNOPSIS	void _stkchk (void);	
DESCRIPTION	_stkchk() is an assembly language routine that is called at the beginning of every function that has been compiled with the stack checking option -bd . It attempts to determine if the current stack is below the base that is initialized by the startup code. It also checks to see if a code word placed at the bottom of the stack has been corrupted. If either event has not occurred, control is returned to the function; otherwise, the _stkover() function is called.	

TYPE	<code>_stkover (C) - Miscellaneous</code>	<i>Amiga</i>
NAME	<code>_stkover</code> - report stack overflow	
SYNOPSIS	<code>void _stkover (void);</code>	
DESCRIPTION	<p><code>_stkover()</code> is called by <code>_stkchk()</code> when a stack overflow has occurred. The routine supplied in the library resets the stack pointer to the top of the stack area and displays the following message on standard output:</p> <p style="padding-left: 40px;">Stack overflow!</p> <p>It then calls the <code>_exit()</code> routine.</p>	

TYPE	strcat (C) - String Handling	ANSI
NAME	strcat - concatenate two strings together	
SYNOPSIS	<pre>#include <string.h> char *strcat(char *dest, const char *src);</pre>	
DESCRIPTION	<p>strcat() appends a copy of the string pointed to by <i>src</i> to the string pointed to by <i>dest</i>, so that the resulting length of <i>dest</i> is strlen (dest) + strlen (src). The first character in <i>src</i> will overwrite the ending null in <i>dest</i>. strcat() will then return a pointer to <i>dest</i>.</p> <p>It is important to note that the area pointed to by <i>dest</i> must be large enough to hold both the strings in <i>dest</i> and <i>src</i> (strlen (dest) + strlen (src) + 1) and it is your responsibility to ensure this.</p>	
SEE ALSO	strncat() , strcpy() , strncpy()	

TYPE	strchr (C) - String Handling	ANSI
NAME	strchr - search for the first occurrence of a character in a string	
SYNOPSIS	<pre>#include <string.h> char *strchr (const char *str, int c);</pre>	
DESCRIPTION	<p>strchr() returns a pointer to the first occurrence of the character <i>c</i> in string <i>str</i>. If <i>c</i> is not contained in <i>str</i>, then strchr() returns a null <i>str</i>.</p> <p>The strchr() function is equivalent to the Aztec function index(), with the exception that strchr() can match a null character, whereas index() cannot.</p>	
SEE ALSO	strrchr() , index() , and rindex()	

TYPE	strcmp (C) - String Handling	ANSI
NAME	strcmp - compare two strings	
SYNOPSIS	<pre>#include <string.h> int strcmp (const char *str1, const char *str2);</pre>	
DESCRIPTION	<p>strcmp() compares the strings pointed to by <i>str1</i> and <i>str2</i>, and determines whether <i>str1</i> is greater than, less than, or equal to <i>str2</i>. Equality is determined in the following manner:</p> <ul style="list-style-type: none">• strcmp() will perform an unsigned comparison on each character in <i>str1</i> and <i>str2</i>, 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 <i>str1</i> is reached but not the end of <i>str2</i>, then <i>str1</i> is less than <i>str2</i>.• If the end of <i>str2</i> is reached before <i>str1</i>, then <i>str1</i> is greater than <i>str2</i>.• 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 <i>str1</i> is greater than <i>str2</i> if <i>str1</i>'s character is greater than <i>str2</i>. Otherwise, <i>str1</i> is less than <i>str2</i>.	

- 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\(\)](#)

TYPE	strcoll (C) - String Handling	<i>ANSI</i>
NAME	strcoll - compare two strings using the current locale	
SYNOPSIS	#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()	

TYPE	strcpy (C) - String Handling	ANSI
NAME	strcpy - copy one string to another	
SYNOPSIS	<pre>#include <string.h> char *strcpy (char *dest, const char *src);</pre>	
DESCRIPTION	strcpy() copies the characters in the string pointed to by <i>src</i> to the area pointed to by <i>dest</i> , including the terminating null character. The area pointed to by <i>dest</i> must be at least (strlen() (<i>src</i>) + 1) characters long. strcpy() always returns <i>dest</i> .	
SEE ALSO	strncpy(), memcpy(), memmove(), strlen()	

TYPE	strcspn (C) - String Handling	ANSI
NAME	strcspn - return the index of the first character in a string <i>str1</i> which is contained in a string <i>str2</i>	
SYNOPSIS	<pre>#include <string.h> size_t strcspn (const char *str1 const char *str2);</pre>	
DESCRIPTION	The strcspn() function returns the index of the first character in the string pointed to by <i>str1</i> which matches a character in the string pointed to by <i>str2</i> . This index is equal to the number of initial characters in <i>str1</i> which do not match a character in <i>str2</i> .	
SEE ALSO	strspn(), strpbrk(), strstr(), strtok()	

TYPE	strlen (C) - String Handling	<i>ANSI</i>
NAME	strlen - return the length of a string	
SYNOPSIS	<pre>#include <string.h> size_t strlen (const char *str);</pre>	
DESCRIPTION	<p>strlen() returns the number of characters in the string pointed to by <i>str</i>, not including the terminating null character.</p> <p>You should be careful in allocating space for strings based on strlen(). A common mistake is to code the following:</p> <pre>ptr = malloc (strlen ("STRING")); strcpy (ptr, "STRING");</pre> <p>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.</p> <p>The correct way to use strlen() is:</p> <pre>ptr = malloc (strlen ("STRING") +1); strcpy (ptr, "STRING");</pre>	
SEE ALSO	strcpy(), strncpy()	

TYPE	strncat (C) - String Handling	ANSI
NAME	strncat - concatenate two strings together, up to <i>max</i> characters	
SYNOPSIS	<pre>#include <string.h> char *strncat(dest, src max);</pre>	
DESCRIPTION	<p>strncat() appends a copy of the string pointed to by <i>src</i> to the string pointed to by <i>dest</i> until either <i>max</i> characters have been appended or a null is reached in <i>src</i>, whichever comes first. The maximum resulting length of the string pointed to by <i>dest</i> will be strlen(<i>dest</i>) and <i>max</i>. The first character in <i>src</i> overwrites the ending null in <i>dest</i>. strncat() always returns a pointer to <i>dest</i>.</p> <p>It is your responsibility to ensure that the area pointed to by <i>dest</i> is at least (strlen(<i>dest</i>) + <i>max</i> + 1) characters long.</p>	
SEE ALSO	strcat() , strcpy() , strncpy()	

TYPE	strncmp (C) - String Handling	ANSI
NAME	strncmp - compare two strings, up to <i>max</i> characters.	
SYNOPSIS	#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 <i>max</i> 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()	

TYPE	strncpy (C) - String Handling	<i>ANSI</i>
NAME	strncpy - copy up to <i>max</i> characters from one string to another	
SYNOPSIS	<pre>#include <string.h> char *strncpy (char *dest, const char *src, size_t max);</pre>	
DESCRIPTION	<p>strncpy() copies a maximum of <i>max</i> characters from the string pointed to by <i>src</i> to the area pointed to by <i>dest</i>. If <i>src</i> is less than <i>max</i> characters long, then <i>dest</i> is padded with null characters until <i>max</i> total characters have been written. The area pointed to by <i>dest</i> must be at least <i>max</i> characters in length.</p>	
SEE ALSO	strcpy(), memcpy(), memmove(), strlen()	

TYPE	strpbrk (C) - String Handling	ANSI
NAME	strpbrk - return the first occurrence in a string <i>str1</i> of a character in a string <i>str2</i> .	
SYNOPSIS	#include <string.h> char *strpbrk (const char * <i>str1</i> , const char * <i>str2</i>);	
DESCRIPTION	The strpbrk() function returns a pointer to the first character in the string pointed to by <i>str1</i> which is contained in the string pointed to by <i>str2</i> . Null is returned if no characters within <i>str1</i> match those in <i>str2</i> .	
SEE ALSO	strspn(), strcspn(), strstr(), strtok()	

TYPE	strrchr (C) - String Handling	<i>ANSI</i>
NAME	strrchr - search for the last occurrence of a character in a string	
SYNOPSIS	<pre>#include <string.h> char *strrchr (const char *str, int c);</pre>	
DESCRIPTION	<p>strrchr() returns the last occurrence of the character <i>c</i> in string <i>str</i>. If <i>c</i> is not contained in <i>str</i>, then strrchr() returns a null.</p> <p>The strrchr() function is equivalent to the Aztec function rindex(), with the exception that strrchr() can match a null character, whereas rindex() cannot.</p>	
SEE ALSO	strchr() , index() , and rindex()	

TYPE	strspn (C) - String Handling	<i>ANSI</i>
NAME	strspn - return the index of the first character in a string <i>str1</i> which is not contained in a string <i>str2</i>	
SYNOPSIS	<pre>#include <string.h> size strspn (const char *str1, const char *str2);</pre>	
DESCRIPTION	The strspn() function returns the index of the first character in <i>str1</i> which is not contained in the string pointed to by <i>str2</i> . This index is equal to the number of initial characters in <i>str1</i> which do match characters in <i>str2</i> .	
SEE ALSO	strcspn(), strpbrk(), strstr(), strtok()	

TYPE	strstr (C) - String Handling	ANSI
NAME	strstr - return a pointer of the first occurrence of a string <i>str2</i> within a string <i>str1</i>	
SYNOPSIS	<pre>#include <string.h> char *strstr(const char *str1, const char *str2);</pre>	
DESCRIPTION	strstr() returns the first occurrence of the substring <i>str2</i> contained within the string <i>str1</i> . If <i>str2</i> is not contained within <i>str1</i> , then null is returned.	
SEE ALSO	stcspn(), strspn(), strpbrk(), strtok()	

TYPE	strtod (C) - Conversion	ANSI
NAME	strtod - convert a string to a double	
SYNOPSIS	<pre>#include <stdlib.h> double strtod (const char *nptr, char **endptr);</pre>	
DESCRIPTION	<p>strtod() converts the initial portion of the string pointed to by <i>nptr</i> into a double value, which is returned by strtod(). The string must be in the form:</p> <p style="text-align: center;"><i>[ws] [+/-] ddd [.] [ddd] [exp]</i></p> <p>where</p> <ul style="list-style-type: none">• <i>ws</i> is whitespace (newline, space, tab)• <i>+/-</i> means <i>+</i> or <i>-</i>• <i>ddd</i> are digits 0-9• <i>exp</i> is an optional exponent of the form: <p style="text-align: center;"><i>[e/E [+/- ddd]]</i></p> <p>Note: In the above input, the slashes indicate "or".</p> <p>strtod stops reading characters when it encounters a character in <i>nptr</i> which cannot be interpreted as part of a double value. It sets <i>*endptr</i> equal to a pointer to this character (as long as <i>endptr</i> is not null).</p>	
DIAGNOSTICS	strtod will return HUGE_VAL if the string causes overflow.	
SEE ALSO	atof() , strtol() , strtol() , atol() , atoi()	

TYPE	strtok (C) - String Handling	ANSI
NAME	strtok - tokenize a string	
SYNOPSIS	<pre>#include <string.h> char *strtok (char *str, const char *del_list);</pre>	
DESCRIPTION	<p>strtok() is designed to be called multiple times to break <i>str</i> into a series of tokens, each of which is delimited by a character contained in <i>del_list</i>. The operation of strtok() is detailed below.</p> <p>First Invocation:</p> <p>On the first invocation of strtok() with <i>str</i>, strtok() will search for the first character in <i>str</i> which is not contained in <i>del_list</i> (that is, it scans past delimiters.)</p> <ul style="list-style-type: none">• If a character is found which is not in <i>del_list</i>, it is considered to be the start of the first token.• If such a character is not found, there are no tokens in <i>str</i> and a null pointer is void. <p>strtok() then searches from the token's start for a character that is contained in <i>del_list</i> (it scans to the next delimiter.)</p> <ul style="list-style-type: none">• If a character is found in <i>str</i> which is contained in <i>del_list</i>, 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 <i>str</i> are found to be in <i>del_list</i>, then the current token is considered to extend to the end of <i>str</i>. 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()**

TYPE	strxfrm (C) - String Handling	ANSI
NAME	strxfrm - transform a string to match the current locale	
SYNOPSIS	<pre>#include <string.h> size_t strxfrm (char *dest, const char *src, size_t n);</pre>	
DESCRIPTION	<p>strxfrm() transforms the string pointed at by <i>src</i> to match the current locale. The transformed string is then copied to <i>dest</i>, and the length of <i>dest</i> is returned.</p> <p>Because only the "c" locale is currently supported, strxfrm() actually does no transformations, but simply performs a strcpy() followed by a strlen().</p>	
SEE ALSO	strcpy(), strcoll(), strcmp(), strlen()	

TYPE	tan (M) - Float Math	ANSI						
NAME	tan - return the tangent of a double value							
SYNOPSIS	#include <math.h> double tan(double x);							
DESCRIPTION	<p>tan() returns the tangent of x. x should be specified in radians.</p> <p>Error codes:</p> <table><thead><tr><th><i>condition</i></th><th><i>return value</i></th><th><i>errno</i></th></tr></thead><tbody><tr><td>$x \geq 6.74652e^9$</td><td>0.0</td><td>ERANGE</td></tr></tbody></table>	<i>condition</i>	<i>return value</i>	<i>errno</i>	$ x \geq 6.74652e^9$	0.0	ERANGE	
<i>condition</i>	<i>return value</i>	<i>errno</i>						
$ x \geq 6.74652e^9$	0.0	ERANGE						

TYPE	tanh (M) - Float Math	ANSI
NAME	tanh - return the hyperbolic tangent of a double value	
SYNOPSIS	<pre>#include <math.h> double tanh (double x);</pre>	
DESCRIPTION	tanh() returns the hyperbolic tangent of <i>x</i> .	

TYPE	time (C) - Time	ANSI
NAME	time - return the time of day	
SYNOPSIS	<pre>#include <time.h> time_t time(time_t *timeptr);</pre>	
DESCRIPTION	The time() function places the current time of day into the location pointed to by <i>timeptr</i> . This value is suitable for use by the ctime() , gmtime() , and localtime() functions. The current time is also returned by time() , in the same format as is placed in <i>timeptr</i> .	
SEE ALSO	ctime() , gmtime() , localtime()	

TYPE	tmpfile (C) - Standard I/O	<i>ANSI</i>
NAME	tmpfile - create a temporary file	
SYNOPSIS	<pre>#include <stdio.h> FILE *tmpfile(void);</pre>	
DESCRIPTION	<p>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.</p> <p>When the temporary file is closed, either because the program explicitly closes it or because the program terminates, the temporary file is automatically deleted.</p> <p>If the file cannot be created, the function returns a null pointer.</p>	
SEE ALSO	tmpnam() , mktemp()	

TYPE	tmpnam (C) - Standard I/O	<i>ANSI</i>
NAME	tmpnam - create a name for a temporary file	
SYNOPSIS	<pre>#include <stdio.h> char *tmpnam (char *s);</pre>	
DESCRIPTION	<p>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.</p> <p><i>s</i> optionally points to an area into which the name will be generated. This must contain at least <i>L_tmpnam</i> bytes, where <i>L_tmpnam</i> is a constant defined in <i>stdio.h</i>.</p> <p><i>s</i> 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.</p> <p>tmpnam() can be called a maximum of <i>TMP_MAX</i> times.</p>	
SEE ALSO	tmpfile() , mktemp()	

TYPE	tolower (C) - Conversion	<i>ANSI</i>
NAME	tolower - convert a character to lowercase	
SYNOPSIS	<pre>#include <ctype.h> int tolower (int c);</pre>	
DESCRIPTION	tolower() converts an uppercase character to lowercase. If the value of <i>c</i> is an uppercase character, tolower() returns its lowercase equivalent, otherwise <i>c</i> is returned unchanged.	
SEE ALSO	toupper()	

TYPE	toupper (C) - Conversion	<i>ANSI</i>
NAME	toupper - convert a character to uppercase	
SYNOPSIS	<pre>#include <ctype.h> int toupper(int c);</pre>	
DESCRIPTION	toupper() converts a lowercase character to uppercase. If the value of the argument <i>c</i> is a lowercase character, toupper() returns its uppercase equivalent as its value, otherwise <i>c</i> is returned unchanged.	
SEE ALSO	tolower()	

TYPE	ungetc (C) - Standard I/O	ANSI
NAME	ungetc - push a character back into input stream	
SYNOPSIS	#include <stdio.h> int ungetc(int <i>c</i> , FILE * <i>stream</i>);	
DESCRIPTION	ungetc() pushes the character value of the argument <i>c</i> back onto an input <i>stream</i> . That character will be returned by the next getc() call on that <i>stream</i> . ungetc() returns <i>c</i> 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.	

TYPE	unlink (C) - UNIX I/O	<i>Aztec/UNIX</i>
NAME	unlink - erase file	
SYNOPSIS	<pre>int unlink (<i>name</i>); char *<i>name</i>;</pre>	
DESCRIPTION	<p>unlink() erases a file, where <i>name</i> is a pointer to a character array containing the name of the file to be erased.</p> <p>unlink() returns a 0 value if successful.</p> <p>Since unlink() is not defined by ANSI, it is strongly recommended that the equivalent ANSI function remove() be used in its place.</p>	
DIAGNOSTICS	<p>unlink() returns -1 if it could not erase the file and places a code in the global integer <i>errno</i> describing the error.</p>	

TYPE	va_arg, va_end, va_start (C) - Variable Arguments	ANSI
NAME	va_arg, va_end, va_start - variable argument access	
SYNOPSIS	<pre>#include <stdarg.h> type va_arg (va_list argptr, type); void va_end (va_list argptr); void va_start (va_list argptr, parm N);</pre>	
DESCRIPTION	<p>These three macros are used as a mechanism to access individual arguments from within a function that accepts a variable number of arguments.</p> <p>A function which receives a variable argument count should declare a variable, <i>argptr</i>, of <i>type va_list</i>. This variable will be used as a pointer to the current argument being processed.</p> <p>The va_start() macro should be called first to initialize <i>argptr</i> to point to the first argument in the list. the <i>parm N</i> parameter should be the last fixed argument passed to the function.</p> <p>Once va_start() has been called, va_arg() may be called repeatedly to fetch arguments sequentially from the argument list. The <i>type</i> parameter to va_arg() should be the C-data <i>type</i> of the next argument (i.e., int, lens, char x, etc.). va_arg() returns the value of the argument.</p> <p>When variable-argument processing is completed, the va_end() macro should be called.</p>	

TYPE	vfprintf (C) - Standard I/O	<i>ANSI</i>
NAME	vfprintf - write formatted ASCII data to a stream	
SYNOPSIS	<pre>#include <stdarg.h> #include <stdio.h> int vfprintf (FILE *stream, const char *fmt, va_list args);</pre>	
DESCRIPTION	<p>vfprintf() is equivalent to the fprintf() function, except that the variable argument list is replaced by <i>args</i>. <i>args</i> should be initialized by the va_start() macro before the call to vfprintf() is made.</p> <p>The return value of vfprintf() is equal to the number of characters written or is a negative value if a write error occurs.</p>	
SEE ALSO	vprintf() , vsprintf() , printf() , va_start()	

TYPE	vprintf (C) - Standard I/O	ANSI
NAME	vprintf - write formatted ASCII data to stdout	
SYNOPSIS	<pre>#include <stdarg.h> #include <stdio.h> int vfprintf (FILE *stream, const char *fmt, va_list args)</pre>	
DESCRIPTION	<p>vprintf() is equivalent to the printf() function, except that the variable argument list is replaced by <i>args</i>. <i>args</i> should be initialized by the va_start() macro before the call to vprintf() is made.</p> <p>The return value of vprintf() is equal to the number of characters written or is a negative value if an error occurred in output.</p>	
SEE ALSO	vfprint(), vsprintf(), printf(), va_start()	

TYPE	vsprintf (C) - Conversion	<i>ANSI</i>
NAME	vsprintf - write formatted ASCII data to a buffer	
SYNOPSIS	<pre>#include <stdarg.h> #include <stdio.h> int vsprintf (char *buf, const char *fmt, va_list args);</pre>	
DESCRIPTION	<p>vsprintf() is equivalent to the sprintf() function, except that the variable argument list is replaced by <i>args</i>. <i>args</i> should be initialized by the va_start() macro before the call to vsprintf() is made.</p> <p>The return value of vsprintf() is the number of characters written, not including the terminating null characters.</p>	
SEE ALSO	vfprintf() , vprintf() , printf() , va_start()	

TYPE	<code>_wb_parse (C) - Amiga</code>	<i>Amiga</i>
NAME	<code>_wb_parse</code> - open standard I/O window	
SYNOPSIS	<pre>void _wb_parse (struct Process *pp, struct WBStartup *wbm);</pre>	
DESCRIPTION	<p><code>_wb_parse()</code> is called from the <code>_main()</code> routine and is used to open a window for standard I/O to use. The window is actually defined by setting the ToolType, WINDOW, to the desired window specification. If this is not required, this routine may be replaced by a stub in your main program. Note that even if this code is called by <code>_main()</code>, if the WINDOW tool type is not defined, there will be no window.</p>	
EXAMPLE	<pre>WINDOW=CON:0/0/640/200/Test Window</pre>	

TYPE	write (C) - UNIX I/O	<i>Aztec /UNIX</i>
NAME	write - write to a file or device using unbuffered I/O	
SYNOPSIS	<code>size_t write(int <i>fd</i>, int <i>bufsize</i>, char *<i>buf</i>);</code>	
DESCRIPTION	<p>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.</p> <p><i>fd</i> is the file descriptor which was returned to the caller when the device or file was opened.</p> <p><i>buf</i> is a pointer to the buffer containing the characters to be written.</p> <p><i>bufsize</i> is the number of characters to be written.</p> <p>If the operation is successful, write() returns as its value the number of characters written.</p>	
SEE ALSO	open() , close() , read()	
DIAGNOSTICS	If the operation is unsuccessful, write() returns -1 and places a code in the global integer errno .	

WORKBENCH

4

Chapter 4 - Workbench

Amiga Reference Guides

This chapter only lists the Amiga Workbench function's parameters, types, and the type of value returned. For detailed information, refer to:

- *Amiga ROM Kernel Manual, Volumes 1 and 2.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga ROM Kernel Reference Manual: Libraries and Devices.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga ROM Kernel Reference Manual: Exec.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga Hardware Reference Manual.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)
- *Amiga Intuition Reference Manual.* (Commodore Business Machines, Inc., Amiga Technical Reference Series, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts)

These manuals should be available through your local bookstore. Refer to the section "Suggestions for Further Reading" at the front of your *Aztec C User Guide* for a listing of other helpful resources.

If you are using Amiga Workbench routines, you should read material relating to the Amiga Workbench functions, the `functions.h` header, and direct resident library calls, in the **Getting Started** Chapter of the *Aztec C User Guide* and the **Compiler** Chapter of the *Aztec C Reference Manual*.

Amiga Workbench Functions

```
void AbortIO(ioRequest)
    struct IORequest *ioRequest;
long ActivateGadget(gadgets, window, req)
    struct Gadgets *gadgets;
    struct Window *window;
    struct Requester *req;
void ActivateWindow(window)
    struct Window *window;
void AddAnimOb(anOb, anKey, rPort)
    struct AnimOb *anOb, **anKey;
    struct RastPort *rPort;
void AddBob(bob, rPort)
    struct Bob *bob;
    struct RastPort *rPort;
void AddConfigDev(configdev)
    struct ConfigDev *configdev;
void AddDevice(device)
    struct Device *device;
long AddDosNode(bootpri, flags,*dosnode)
    long bootpri, flags;
    struct DeviceNode *dosnode;
void AddFont(textFont)
    struct TextFont *textFont;
long AddFreeList(free, mem, len)
    struct FreeList *free;
    char *mem;
    long len;
```

```
long AddGadget(pntr, gadget, position)
    struct Window *pntr;
    struct Gadget *gadget;
    long position;
long AddGList(addptr, gadget, pos, numgad, req)
    struct Window *addptr;
    struct Gadget *gadget;
    long pos, numgad;
    struct Requester *req;
void AddHead(list, node)
    struct List *list;
    struct Node *node;
void AddIntServer(intNum, interrupt)
    long intNum;
    struct Interrupt *interrupt;
void AddLibrary(library)
    struct Library *library;
long AddMemList(size, attrib, pri, base, name)
    long size, attrib, pri;
    void *base;
    char *name;
void AddPort(port)
    struct MsgPort *port;
void AddResource(resource)
    struct MiscResource *resource;
void AddSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void AddTail(list, node)
    struct List *list;
    struct Node *node;
void AddTask(task, initialPC, finalPC)
    struct Task *task;
    void *initialPC, *finalPC;
```

```
void AddTime(dest, source)
    struct timeval *dest, *source;
void AddVSprite(vs, rPort)
    struct VSprite *vs;
    struct RastPort *rPort;
long Alert(alertNum, parameters)
    long alertNum, parameters;
void *AllocAbs(bytSiz, location)
    long bytSiz;
    void *location;
long AllocBoardMem(slotSpec)
    long slotSpec;
long AllocCList(cLPool)
    long cLPool;
struct ConfigDev *AllocConfigDev()
struct MemList *AllocEntry(memList)
    struct MemList *memList;
long AllocExpansionMem(numSlots, SAlign, SOffset)
    long numSlots, SAlign, SOffset;
void *AllocMem(byteSize, requirements)
    long byteSize, requirements;
long AllocPotBits(bits)
    long bits;
void AllocRaster(width, height)
    long width, height;
void *AllocRemember(rememberKey, size, flags)
    struct Remember **rememberKey;
    long size, flags;
long AllocSignal(signalNum)
    long signalNum;
long AllocTrap(trapNum)
    long trapNum;
struct WObject *AllocWObject()
```

```
void *Allocate(freeList, byteSize)
    struct MemHeader *freeList;
    long byteSize;
void AlohaWorkbench(wbPort)
    struct MsgPort *wbPort;
void AndRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
long AndRegionRegion(src, dst)
    struct Region *src, *dst;
void Animate(key, rPort)
    struct AnimOb **key;
    struct RastPort *rPort;
long AreaCircle(rp, cx, cy, r)
    struct RastPort *rp;
    long cx, cy, r;
short AreaDraw(rp, x, y)
    struct RastPort *rp;
    long x, y;
long AreaEllipse(rp, cx, cy, a, b)
    struct RastPort *rp;
    long cx, cy, a, b;
void AreaEnd(rp)
    struct RastPort *rp;
long AreaMove(rp, x, y)
    struct RastPort *rp;
    long x, y;
void AskFont(rp, textAttr)
    struct RastPort *rp;
    struct TextAttr *textAttr;
long AskSoftStyle(rp)
    struct RastPort *rp;
```

```
long AttemptLockLayerRom(layer)
    struct Layer *layer;
long AttemptSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
long AutoRequest(window, body, positive, negative, posFlags,
    negFlags, width, height)
    struct Window *window;
    struct IntuiText *body, *positive, *negative;
    long posFlags, negFlags, width, height;
long AvailFonts(buffer, bufBytes, types)
    char *buffer;
    long bufBytes, types;
long AvailMem(requirements)
    long requirements;
void BeginIO(ioRequest)
    struct IORequest *ioRequest;
void BeginRefresh(window)
    struct Window *window;
long BeginUpdate(layer)
    struct Layer *layer;
long BehindLayer(li, layer)
    struct Layer_Info *li;
    struct Layer *layer;
long BltBitMap(srcBM, srcX, srcY, dstBM, dstX, dstY, sizX, sizY,
    minTerm, mask, tempA)
    struct BitMap *srcBM, *dstBM;
    long srcX, srcY, dstX, dstY, sizX, sizY, minTerm, mask;
    char *tempA;
long BltBitMapRastPort(srcBitMap, srcX, srcY, dstRPort,
    dstX, dstY, sizX, sizY, minTerm)
    struct BitMap *srcBitMap;
    struct RastPort *dstRPort;
    long srcX, srcY, dstX, dstY, sizX, sizY, minTerm;
```

```
void BltClear(memBlock, byteCount, flags)
    char *memBlock;
    long byteCount, flags;
long BltMaskBitMapRastPort(srcBitMap, srcX, srcY, dstRPort,
    dstX, dstY, sizX, sizY, minTerm, bltmask)
    struct BitMap *srcBitMap ;
    struct RastPort *dstRPort
    long srcX, srcY, dstX, dstY, sizX, sizY, minTerm;
    void *bltmask;
void BltPattern(rp, buf, x1, y1, maxX, maxY, byteCnt)
    struct RastPort *rp;
    char *buf;
    long x1, y1, maxX, maxY, byteCnt;
void BltTemplate(src, srcX, srcMod, dstRastPort, dstX, dstY,
    sizX, sizY)
    char *src;
    struct RastPort *dstRastPort;
    long srcX, srcMod, dstX, dstY, sizX, sizY;
void BNDRYOFF(rp)
    struct RastPort *rp;
struct Window *BuildSysRequest(wind, body, positive,
    negative, flags, width, height)
    struct Window *wind;
    struct IntuiText *body, *positive, *negative;
    long flags, width, height;
char *BumpRevision(newbuf, oldname)
    char *newbuf, *oldname;
void CBump(copperlist)
    struct UCopList *copperlist;
struct InputEvent *CDInputHandler(events, consoleDevice)
    struct InputEvent *events;
    struct Device *consoleDevice;
```

```
void CEND(c)
    struct UCopList *c;
long Chk_Abort()
void CINIT(c, n)
    struct UCoplist *c;
    long n;
void CMove(c, a, v)
    struct UCopList *c;
    long *a, v;
void CWait(c, v, h)
    struct UCopList *c;
    long v, h;
void Cause(interrupt)
    struct Interrupt *interrupt;
void ChangeSprite(vp, s, newdata)
    struct ViewPort *vp;
    struct SimpleSprite *s;
    short *newdata;
struct IORequest *CheckIO(ioRequest)
    struct IORequest *ioRequest;
long ClearDMRequest(window)
    struct Window *window;
void ClearEOL(rp)
    struct RastPort *rp;
void ClearMenuStrip(window)
    struct Window *window;
void ClearPointer(window)
    struct Window *window;
long ClearRectRegion(rgn, rect)
    struct Region *rgn;
    struct Rectangle *rect;
void ClearRegion(region)
    struct Region *region;
```

```
void ClearScreen(rp)
    struct RastPort *rp;
void ClipBlit(src, srcX, srcY, dst, dstX, dstY, xSize, ySize, mode)
    struct RastPort *src, *dst;
    long srcX, srcY, dstX, dstY, xSize, ySize, mode;
void Close(file)
    struct FileHandle *file;
void CloseDevice(ioRequest)
    struct IORequest *ioRequest;
void CloseFont(font)
    struct TextFont *font;
void CloseLibrary(library)
    void *library;
void CloseScreen(screen)
    struct Screen *screen;
void CloseWindow(window)
    struct Window *window;
long CloseWorkBench()
void CMOVE(c, a, b)
    struct UCopList *c;
    long a, b;
long CmpTime(dest, source)
    struct timeval *dest, *source;
long ConcatCList(sourceCList, destCList)
    long sourceCList, destCList;
long ConfigBoard(board, configDev)
    long board;
    struct ConfigDev *configDev;
long ConfigChain(baseAddr)
    long baseAddr;
long CopyCList(cList)
    long cList;
```

```
void CopyMem(src, dest, size)
    char *src, *dest;
    long size;
void CopyMemQuick(src, dest, size)
    char *src, *dest;
    long size;
void CopySBitMap(layer)
    struct Layer *layer;
struct Layer *CreateBehindLayer(li, bm, x0, y0, x1, y1, flags, bm2)
    struct Layer_Info *li;
    struct BitMap *bm, *bm2;
    long x0, y0, x1, y1, flags;
struct FileLock *CreateDir(name)
    char *name;
struct IORequest *CreateExtIO(iorpt, size)
    struct MsgPort *iorpt;
    long size;
struct MsgPort *CreatePort(name, pri)
    char *name;
    long pri;
struct Process *CreateProc(name, pri, segment, stackSize)
    char *name;
    long pri, stackSize, segment;
struct IOStdReq *CreateStdIO(mp)
    struct MsgPort *mp;
struct Task *CreateTask(name, pri, start_pc, stksiz)
    char *name;
    long pri, stksiz;
    void *start_pc;
struct Layer *CreateUpfrontLayer(li, bm, x0, y0, x1, y1, flags, bm2)
    struct Layer_Info *li;
    struct BitMap *bm, *bm2;
    long x0, y0, x1, y1, flags;
```

```
struct FileLock *CurrentDir(lock)
    struct FileLock *lock;
void CurrentTime(seconds, micros)
    long *seconds, *micros;
void CWAIT(c, a, b)
    struct UCopList *c;
    long a, b;
long *DateStamp(v)
    long *v;
void Deallocate(freeList, memoryBlock, byteSize)
    struct MemHeader *freeList;
    void *memoryBlock;
    long byteSize;
void Debug()
void Delay(timeout)
    long timeout;
void DeleteExtIO(ioreq)
    struct IORequest *ioreq;
long DeleteFile(name)
    char *name;
void DeleteLayer(li, l)
    struct Layer_Info *li;
    struct Layer *l;
void DeletePort(port)
    struct MsgPort *port;
void DeleteStdIO(iop)
    struct IOStdReq *iop;
void DeleteTask(tp)
    struct Task *tp;
struct Process *DeviceProc(name)
    char *name;
void Disable()
void DisownBlitter()
void Dispatch()
```

```
void DisplayAlert(alertNumber, string, height)
    long alertNumber;
    char *string;
    long height;
void DisplayBeep(screen)
    struct Screen *screen;
void DisposeLayerInfo(li)
    struct Layer_Info *li;
void DisposeRegion(region)
    struct Region *region;
void DoCollision(rPort)
    struct RastPort *rPort;
long DoIO(ioRequest)
    struct IORequest *ioRequest;
long DoubleClick(startSeconds, startMicros, currentSeconds,
                 currentMicros)
    long startSeconds, startMicros,
         currentSeconds, currentMicros;
void Draw(rp, x, y)
    struct RastPort *rp;
    long x, y;
void DrawBorder(rp, b, leftOffset, topOffset)
    struct RastPort *rp;
    struct Border *b;
    long leftOffset, topOffset;
void DrawCircle(rp, cx, cy, r)
    struct RastPort *rp;
    long cx, cy, r;
void DrawEllipse(rp, cx, cy, a, b)
    struct RastPort *rp;
    long cx, cy, a, b;
void DrawGList(rPort, vPort)
    struct RastPort *rPort;
    struct ViewPort *vPort;
```

```
void DrawImage(rp, image, leftOffset, topOffset)
    struct RastPort *rp;
    struct Image *image;
    long leftOffset, topOffset;
struct FileLock *DupLock(lock)
    struct FileLock *lock;
void Enable()
void EndRefresh(window, complete)
    struct Window *window;
    long complete;
void EndRequest(req, wind)
    struct Requester *req;
    struct Window *wind;
void EndUpdate(l, flag)
    struct Layer *l;
    long flag;
void Enqueue(list, node)
    struct List *list;
    struct Node *node;
void Exception()
void ExitIntr()
long ExNext(lock, fileInfoBlock)
    struct FileLock *lock;
    struct FileInfoBlock *fileInfoBlock;
long Examine(lock, fileInfoBlock)
    struct FileLock *lock;
    struct FileInfoBlock *fileInfoBlock;
long Execute(commandString, input, output)
    char *commandString;
    struct FileHandle *input, *output;
void Exit(returnCode)
    long returnCode;
void FattenLayerInfo(li)
    struct Layer_Info *li;
```

```
struct ConfigDev *FindConfigDev(oldCnfdev, manu, prod)
    struct ConfigDev *oldCnfdev;
    long manu, prod;
struct Node *FindName(list, name)
    struct List *list;
    char *name;
struct MsgPort *FindPort(name)
    char *name;
struct Resident *FindResident(name)
    char *name;
struct SignalSemaphore *FindSemaphore(name)
    char *name;
struct Task *FindTask(name)
    char *name;
char *FindToolType(toolTypeArray, typeName)
    char **toolTypeArray, *typeName;
long Flood(rp, mode, x, y)
    struct RastPort *rp;
    long mode, x, y;
void FlushCList(cList)
    long cList;
void Forbid()
void FreeBoardMem(stslot, stspec)
    long stslot, stspec;
void FreeCList(cList)
    long cList;
void FreeColorMap(colorMap)
    struct ColorMap *colorMap;
void FreeConfigDev(configdev)
    struct ConfigDev *configdev;
void FreeCopList(copList)
    struct CopList *copList;
void FreeCprList(cprList)
    struct cprlist *cprList;
```

```
void FreeDiskObject(diskobj)
    struct DiskObject *diskobj;
void FreeEntry(memList)
    struct MemList *memList;
void FreeExpansionMem(stslot, numslots)
    long stslot, numslots;
void FreeFreeList(free)
    struct FreeList *free;
void FreeGBuffers(anOb, RPort, db)
    struct AnimOb *anOb;
    struct RastPort *RPort;
    long db;
void FreeMem(memoryBlock, sizeBytes)
    void *memoryBlock;
    long sizeBytes;
void FreePotBits(allocated)
    long allocated;
void FreeRaster(p, width, height)
    void *p;
    long width, height;
void FreeRemember(key, reallyForget)
    struct Remember **key;
    long reallyForget;
void FreeSignal(signalNum)
    long signalNum;
void FreeSprite(pick)
    long pick;
void FreeSysRequest(wind)
    struct Window *wind;
void FreeTrap(trapNum)
    long trapNum;
void FreeVPortCopLists(viewPort)
    struct ViewPort *viewPort;
```

```
void FreeWBObject(obj)
    struct WBObject *obj;
long GetCC()
long GetCLBuf(cList, buffer, maxlen)
    long cList;
    char *buffer;
    long maxlen;
long GetCLChar(cList)
    long cList;
long GetCLWord(cList)
    long cList;
struct ColorMap *GetColorMap(entries)
    long entries;
long GetCurrentBinding(currbd, bsize)
    struct CurrentBinding *currbd;
    long bsize;
struct Preferences *GetDefPrefs(prefBuffer, size)
    struct Preferences *prefBuffer;
    long size;
struct DiskObject *GetDiskObject(name)
    char *name;
long GetGBuffers(anOb, rPort, db)
    struct AnimOb *anOb;
    struct RastPort *rPort;
    long db;
long GetIcon(name, icon, free)
    char *name;
    struct DiskObject *icon;
    struct FreeList *free;
struct Message *GetMsg(port)
    struct MsgPort *port;
long GetPacket(wait)
    long wait;
```

```
struct Preferences *GetPrefs(buffer, size)
    struct Preferences *buffer;
    long size;
long GetRGB4(colorMap, entry)
    struct ColorMap *colorMap;
    long entry;
long GetScreenData(buff, size, type, screen)
    char *buff;
    long size, type;
    struct Screen *screen;
long GetSprite(sprite, pick)
    struct SimpleSprite *sprite;
    long pick;
struct WBOBJECT *GetWBOBJECT(name)
    char *name;
void GraphicsReserved1()
void GraphicsReserved2()
long IncrCLMark(cList)
    long cList;
long Info(lock, infoData)
    struct FileLock *lock;
    struct InfoData *infoData;
void InitAnimate(akey)
    struct AnimOb **akey;
void InitArea(areaInfo, buffer, maxVectors)
    struct AreaInfo *areaInfo;
    short *buffer;
    long maxVectors;
void InitBitMap(bm, depth, width, height)
    struct BitMap *bm;
    long depth, width, height;
long InitCLPool(cLPool, size)
    struct cLPool *cLPool;
    long size;
```

```
void InitCode(startClass, version)
    long startClass, version;
void InitGMasks(anOb)
    struct AnimOb *anOb;
void InitGels(head, tail, gInfo)
    struct VSprite *head;
    struct VSprite *tail;
    struct GelsInfo *gInfo;
void InitLayers(li)
    struct Layer_Info *li;
void InitMasks(vs)
    struct VSprite *vs;
void InitRastPort(rp)
    struct RastPort *rp;
void InitRequester(req)
    struct Requester *req;
void InitResident(resident, segList)
    struct Resident *resident,
    long segList;
void InitSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void InitStruct(initTable, memory, size)
    short *initTable;
    void *memory;
    long size;
void InitTmpRas(tmpRas, buffer, size)
    struct TmpRas *tmpRas;
    char *buffer;
    long size;
void InitVPort(vp)
    struct ViewPort *vp;
void InitView(view)
    struct View *view;
```

```
struct FileHandle *Input()
void Insert(list, node, listNode)
    struct List *list;
    struct Node *node, *listNode;
struct Region *InstallClipRegion(layer, region)
    struct Layer *layer;
    struct Region *region;
long IntuiTextLength(iText)
    struct IntuiText *iText;
struct InputEvent *Intuition(inputEvent)
    struct InputEvent *inputEvent;
long IoErr()
long IsInteractive(file)
    struct FileHandle *file;
struct MenuItem *ItemAddress(menuStrip, menuNumber)
    struct Menu *menuStrip;
    long menuNumber;
void LoadRGB4(vp, colorMap, count)
    struct ViewPort *vp;
    unsigned short *colorMap;
    long count;
long *LoadSeg(name)
    char *name;
void LoadView(view)
    struct View *view;
struct FileLock *Lock(name, accessMode)
    char *name;
    long accessMode;
long LockIBase(dontknow)
    long dontknow;
void LockLayer(li, l)
    struct Layer_Info *li;
    struct Layer *l;
```

```
void LockLayerInfo(li)
    struct Layer_Info *li;
void LockLayerRom(layer)
    struct Layer *layer;
void LockLayers(li)
    struct Layer_Info *li;
struct DeviceNode *MakeDosNode(parmpacket)
    long *parmpacket;
long MakeFunctions(target, functionarray, dispbase)
    char *target, *dispbase;
    void(*functionarray[])();
struct Library *MakeLibrary(vectors, structure, libInit,
    dataSize, segList)
    void(*vectors[])(),(*libInit)();
    short *structure;
    long dataSize;
    struct MemList *segList;
void MakeScreen(screen)
    struct Screen *screen;
void MakeVPort(view, viewPort)
    struct View *view;
    struct ViewPort *viewPort;
long MarkCList(cList, offset)
    long cList;
    long offset;
long MatchToolValue(typeString, value)
    char *typeString, *value;
void ModifyIDCMP(wind, IDCMPFlags)
    struct Window *wind;
    long IDCMPFlags;
```

```
void ModifyProp(gad, wind, req, flags, horizPot, vertPot,
    horizBody, verBody)
    struct Gadget *gad;
    struct Window *wind;
    struct Requester *req;
    long flags, horizPot, vertPot, horizBody, verBody;
void Move(rp, x, y)
    struct RastPort *rp;
    long x, y;
long MoveLayer(li, l, dx, dy)
    struct Layer_Info *li;
    struct Layer *l;
    long dx, dy;
long MoveLayerInFrontOf(layer, linf)
    struct Layer *layer, *linf;
void MoveScreen(screen, deltaX, deltaY)
    struct Screen *screen;
    long deltaX, deltaY;
void MoveSprite(vp, sprite, x, y)
    struct ViewPort *vp;
    struct SimpleSprite *sprite;
    long x, y;
void MoveWindow(wind, deltaX, deltaY)
    struct Window *wind;
    long deltaX, deltaY;
void MrgCop(view)
    struct View *view;
struct Layer_Info *NewLayerInfo()
void NewList(list)
    struct List *list;
```

```
void NewModifyProp(gadg, ptr, req, flags, hp, vp, hb, vb, numgad)
    struct Gadget *gadg;
    struct Window *ptr;
    struct Requester *req;
    long hp, vp, hb, vb, numgad, flags;
struct Region *NewRegion()
void ObtainConfigBinding()
void ObtainSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void ObtainSemaphoreList(sslist)
    struct List *sslist;
void OffGadget(gad, wind, req)
    struct Gadget *gad;
    struct Window *wind;
    struct Requester *req;
void OffMenu(wind, menuNumber)
    struct Window *wind;
    long menuNumber;
void *OldOpenLibrary(libName)
    char *libName;
void OnGadget(gad, wind, req)
    struct Gadget *gad;
    struct Window *wind;
    struct Requester *req;
void OnMenu(wind, menuNumber)
    struct Window *wind;
    long menuNumber;
struct FileHandle *Open(name, accessMode)
    char *name;
    long accessMode;
long OpenDevice(name, unitNumber, ioRequest, flags)
    char *name;
    struct IORequest *ioRequest;
    long unitNumber, flags;
```

```
struct TextFont *OpenDiskFont(textAttr)
    struct TextAttr *textAttr;
struct TextFont *OpenFont(textAttr)
    struct TextAttr *textAttr;
void OpenIntuition()
void *OpenLibrary(libName, version)
    char *libName;
    long version;
struct MiscResource *OpenResource(resName, ver)
    char *resName;
    long ver;
struct Screen *OpenScreen(newScreen)
    struct NewScreen *newScreen;
struct Window *OpenWindow(newWindow)
    struct NewWindow *newWindow;
long OpenWorkBench()
void OrRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
long OrRegionRegion(src, dst)
    struct Region *src, *dst;
struct FileHandle *Output()
void OwnBlitter()
struct FileLock *ParentDir(lock)
    struct FileLock *lock;
long PeekCLMark(cList)
    long cList;
void Permit()
void PolyDraw(rp, count, array)
    struct RastPort *rp;
    long count;
    short *array;
```

```
void PrintIText(rp, iText, leftEdge, topEdge)
    struct RastPort *rp;
    struct IntuiText *iText;
    long leftEdge, topEdge;
long Procure(semaport, bidMsg)
    struct Semaphore *semaport;
    struct Message *bidMsg;
long PutCLBuf(cList, buffer, length)
    long cList;
    char *buffer;
    long length;
long PutCLChar(cList, byte)
    long cList;
    long byte;
long PutCLWord(cList, word)
    long cList;
    long word;
long PutDiskObject(name, diskobj)
    char *name;
    struct DiskObject *diskobj;
long PutIcon(name, icon)
    char *name;
    struct DiskObject *icon;
void PutMsg(port, message)
    struct MsgPort *port;
    struct Message *message;
long PutWBOject(name, object)
    char *name;
    struct WBOBJECT *object;
void QBSBlit(bsp)
    struct bltnode *bsp;
void QBlit(bp)
    struct bltnode *bp;
```

```
long QueuePacket(packet)
    long *packet;
void RawDoFmt(fstr, dstrm, putchp, putchd)
    char *fstr, *sdstrm;
    PVF putchp;
    long putchd;
void RawIOInit();
long RawKeyConvert(events, buffer, length, keyMap)
    struct InputEvent *events;
    char *buffer;
    long length;
    struct KeyMap *keyMap;
long RawMayGetChar()
void RawPutChar(c)
    long c;
long Read(file, buffer, length)
    struct FileHandle *file;
    char *buffer;
    long length;
long ReadExpansionByte(board, offset)
    char *board;
    long offset;
long ReadExpansionRom(board, cdev)
    char *board;
    struct ConfigDev *cdev;
long ReadPixel(rp, x, y)
    struct RastPort *rp;
    long x, y;
void RectFill(rp, xMin, yMin, xMax, yMax)
    struct RastPort *rp;
    long xMin, yMin, xMax, yMax;
```

```
void RefreshGadgets(gad, wind, req)
    struct Gadgets *gad;
    struct Window *wind;
    struct Requester *req;
void RefreshGList(gadg, ptr, req, numgad)
    struct Gadget *gadg;
    struct Window *ptr;
    struct Requester *req;
    long numgad;
void RefreshWindowFrame(wind)
    struct Window *wind;
void ReleaseConfigBinding()
void ReleaseSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
void ReleaseSemaphoreList(sigsem)
    struct List *sigsem;
void RemBob(bob)
    struct Bob *bob;
void RemConfigDev(cdev)
    struct ConfigDev *cdev;
long RemDevice(device)
    struct Device *device;
long RemFont(textFont)
    struct TextFont *textFont;
struct Node *RemHead(list)
    struct List *list;
void RemIBob(bob, rPort, vPort)
    struct Bob *bob;
    struct RastPort *rPort;
    struct ViewPort *vPort;
void RemIntServer(intNum, interrupt)
    long intNum;
    struct Interrupt *interrupt;
```

```
long RemLibrary(library)
    struct Library *library;
long RemoveGList(remptr, gadg, numgad)
    struct Window *remptr;
    struct Gadget *gadg;
    long numgad;
void RemPort(port)
    struct MsgPort *port;
void RemResource(resource)
    struct MiscResource *resource;
void RemSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
struct Node *RemTail(list)
    struct List *list;
void RemTask(task)
    struct Task *task;
void RemVSprite(vs)
    struct VSprite *vs;
void RemakeDisplay()
void Remove(node)
    struct Node *node;
long RemoveGadget(wind, gad)
    struct Window *wind;
    struct Gadget *gad;
long Rename(oldName, newName)
    char *oldName, *newName;
void ReplyMsg(message)
    struct Message *message;
void ReportMouse(boolean, wind)
    struct Window *wind;
    long boolean;
long Request(req, wind)
    struct Requester *req;
    struct Window *wind;
```

```
void Reschedule()
void RethinkDisplay()
void Schedule()
void ScreenToBack(screen)
    struct Screen *screen;
void ScreenToFront(screen)
    struct Screen *screen;
void ScrollLayer(li, l, dx, dy)
    struct Layer_Info *li;
    struct Layer *l;
    long dx, dy;
void ScrollRaster(rp, dx, dy, xMin, yMin, xMax, yMax)
    struct RastPort *rp;
    long dx, dy, xMin, yMin, xMax, yMax;
void ScrollVPort(vp)
    struct ViewPort *vp;
long Seek(file, position, mode)
    struct FileHandle *file;
    long position, mode;
void SendIO(ioRequest)
    struct IORequest *ioRequest;
void SetAfPt(rp, pattern, size)
    struct RastPort *rp;
    unsigned short *pattern;
    long size;
void SetAPen(rp, pen)
    struct RastPort *rp;
    long pen;
void SetBPen(rp, pen)
    struct RastPort *rp;
    long pen;
```

```
void SetCollision(num, routine, gInfo)
    long num;
    void (*routine)();
    struct GelsInfo *gInfo;
long SetComment(name, comment)
    char *name, *comment;
void SetCurrentBinding(currb, bsize)
    struct CurrentBinding *currb;
    long bsize;
long SetDMRequest(wind, DMRequester)
    struct Window *wind;
    struct Requester *DMRequester;
void SetDrMd(rp, mode)
    struct RastPort *rp;
    long mode;
void SetDrPt(rp, pattern)
    struct RastPort *rp;
    long pattern;
long SetExcept(newSignals, signalMask)
    long newSignals, signalMask;
void SetFont(rp, font)
    struct RastPort *rp;
    struct TextFont *font;
void *SetFunction(library, funcOffset, funcEntry)
    struct Library *library;
    long funcOffset;
    void (*funcEntry)();
struct Interrupt *SetIntVector(intNumber, interrupt)
    long intNumber;
    struct Interrupt *interrupt;
void SetMenuStrip(wind, menu)
    struct Window *wind;
    struct Menu *menu;
```

```
void SetOPen(rp, pen)
    struct RastPort *rp;
    long pen;
void SetPointer(wind, sp, height, width, xOffset, yOffset)
    struct Window *wind;
    short *sp;
    long height, width, xOffset, yOffset;
struct Preferences *SetPrefs(p, size, realThing)
    struct Preferences *p;
    long size, realThing;
long SetProtection(name, mask)
    char *name;
    long mask;
void SetRGB4(vp, n, r, g, b)
    struct ViewPort *vp;
    long n, r, g, b;
void SetRGB4CM(cm, i, r, g, b)
    struct ColorMap *cm;
    long i, r, g, b;
void SetRast(rp, pen)
    struct RastPort *rp;
    long pen;
long SetSR(newSR, mask)
    long newSR, mask;
long SetSignal(newSignals, signalMask)
    long newSignals, signalMask;
long SetSoftStyle(rp, style, enable)
    struct RastPort *rp;
    long style, enable;
long SetTaskPri(task, priority)
    struct Task *task;
    long priority;
```

```
void SetWindowTitles(wind, windowTitle, screenTitle)
    struct Window *wind;
    char *windowTitle, *screenTitle;
void SetWrMsk(rp, mask)
    struct RastPort *rp;
    long mask;
void ShowTitle(screen, showIt)
    struct Screen *screen;
    long showIt;
void Signal(task, signals)
    struct Task *task;
    long signals;
long SizeCList(cList)
    long cList;
long SizeLayer(li, l, dx, dy)
    struct Layer_Info *li;
    struct Layer *l;
    long dx, dy;
void SizeWindow(wind, deltaX, deltaY)
    struct Window *wind;
    long deltaX, deltaY;
void SortGList(rPort)
    struct RastPort *rPort;
long SplitCList(cList)
    long cList;
long SubCList(cList, index, length)
    long cList;
    long index, length;
void SubTime(dest, source)
    struct timeval *dest, *source;
void SumKickData()
void SumLibrary(library)
    struct Library *library;
long SuperState()
```

```
void Supervisor()
void SwapBitsRastPortClipRect(rp, cr)
    struct ClipRect *cr;
    struct RastPort *rp;
void Switch()
void SyncSBitMap(layer)
    struct Layer *layer;
long Text(rp, string, count)
    struct RastPort *rp;
    char *string;
    long count;
long TextLength(rp, string, count)
    struct RastPort *rp;
    char *string;
    long count;
void ThinLayerInfo(li)
    struct Layer_Info *li;
long Translate(instring, inlen, outbuf, outlen)
    char *instring, *outbuf;
    long inlen, outlen;
long TypeOfMem(addr)
    char *addr;
void UCopperListInit(copplist, num)
    struct UCopList *copplist;
    long num;
long UnGetCLChar(cList, byte)
    long cList;
    long byte;
long UnGetCLWord(cList, word)
    long cList;
    long word;
long UnLoadSeg(segment)
    long segment;
```

```
void UnLock(lock)
    struct FileLock *lock;
void UnLockIBase(Iblock)
    long Iblock;
long UnPutCLChar(cList)
    long cList;
long UnPutCLWord(cList)
    long cList;
void UnlockLayer(l)
    struct Layer *l;
void UnlockLayerInfo(li)
    struct Layer_Info *li;
void UnlockLayerRom(layer)
    struct Layer *layer;
void UnlockLayers(li)
    struct Layer_Info *li;
long UpfrontLayer(li, l)
    struct Layer_Info *li;
    struct Layer *l;
void UserState(sysStack)
    void *sysStack;
void Vacate(semaport)
    struct Semaphore *semaport;
long VBeamPos()
struct View *ViewAddress()
struct View *ViewPortAddress(wind)
    struct Window *wind;
long WBenchToBack()
long WBenchToFront()
long Wait(signalSet)
    long signalSet;
void WaitBOVP(viewPort)
    struct ViewPort *viewPort;
void WaitBlit()
```

```
long WaitForChar(file, timeout)
    struct FileHandle *file;
    long timeout;
long WaitIO(ioRequest)
    struct IORequest *ioRequest;
struct Message *WaitPort(port)
    struct MsgPort *port;
void WaitTOF()
struct Layer *WhichLayer(li, x, y)
    struct Layer_Info *li;
    long x, y;
long WindowLimits(wind, minWidth, minHeight, maxWidth,
    maxHeight)
    struct Window *wind;
    long minWidth, minHeight, maxWidth, maxHeight;
void WindowToBack(wind)
    struct Window *wind;
void WindowToFront(wind)
    struct Window *wind;
long Write(file, buffer, length)
    struct FileHandle *file;
    char *buffer;
    long length;
long WriteExpansionByte(board, offset, byte)
    char *board;
    long offset, byte;
void WritePixel(rp, x, y)
    struct RastPort *rp;
    long x, y;
void WritePotgo(word, mask)
    long word, mask;
void XorRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
long XorRegionRegion(src, dst)
    struct Region *src, *dst;
```

Index

`_cli_parse`, 3-26
`_exit`, 3-39
`_stkchk`, 3-161
`_stkover`, 3-162
`_wb_parse`, 3-195

A

abort, 3-9
abort tasks, 3-9
aborting programs, 2-8
abs, 3-10
access, 2-15
 errno, 3-12
 mode, 3-11
accessing devices
 unbuffered I/O, 2-3
 standard I/O, 2-3
accessing files, 2-10
 unbuffered I/O, 2-3
 standard I/O, 2-3
acos, 3-13
allocate memory, 3-99
allocate memory space, 3-135
allocate space from system memory, 3-23
Amiga functions
 `_cli_parse`, 3-26
 `_wb_parse`, 3-195
 `dos_packet`, 3-36
 `execl,execl,execv,execvp`, 3-37
 `fexecl`, 3-48
 `fexecv`, 3-48
 `freeseg`, 3-67
 `geta4`, 3-78
 `int_end`, 3-85
 `int_start`, 3-86

`scdir`, 3-141
`scr_beep`, 3-142
`scr_bs`, 3-142
`scr_cdelete`, 3-142
`scr_cinsert`, 3-142
`scr_clear`, 3-142
`scr_cr`, 3-142
`scr_curs`, 3-142
`scr_cursrt`, 3-142
`scr_cursup`, 3-142
`scr_eol`, 3-142
`scr_home`, 3-142
`scr_ldelete`, 3-142
`scr_lf`, 3-142
`scr_linsert`, 3-142
`scr_tab`, 3-142
`segload`, 3-144

Amiga I/O functions, 2-7

Amiga machine dependent functions, 3-87, 3-160

See also Amiga functions

- `_abort`, 3-9
- `_stkchk`, 3-161
- `_stkover`, 3-162
- `mktemp`, 3-105
- `setenv`, 3-146

Amiga Rom Kernel Reference Manual, 2-7

Amiga standard I/O, 2-14

- buffer size, 2-14

amigados, 2-7 - 2-8

ANSI, 1-1, 2-7

- standard I/O functions, 2-3

ANSI functions

- `abs`, 3-10
- `acos`, 3-13
- `asctime`, 3-14
- `asin`, 3-15
- `assert`, 3-16
- `atan`, 3-17
- `atan2`, 3-18
- `atexit`, 3-19

atof, 3-20
atoi, 3-21
atol, 3-22
calloc, 3-23
ceil, 3-24
clearerr, 3-25
clock, 3-27
cos, 3-29
cosh, 3-30
ctime, 3-33
difftime, 3-34
div, 3-35
exit, 3-39
exp, 3-40
fabs, 3-41
fclose, 3-42
feof, 3-46
ferror, 3-47
fflush, 3-51
fgetc, 3-52
fgetpos, 3-53
fgets, 3-54
floor, 3-56
fmod, 3-57
fopen, 3-58
fprintf, 3-62
fputc, 3-63
fputs, 3-64
fread, 3-65
free, 3-66
freopen, 3-68
frexp, 3-69
fscanf, 3-70
fseek, 3-71
fsetpos, 3-73
ftell, 3-74
fwrite, 3-76
getc, 3-79
getchar, 3-80
getenv, 3-81

gets, 3-82
gmtime, 3-83
isalnum, 3-88
isalpha, 3-88
isascii, 3-88
iscntrl, 3-88
isdigit, 3-88
isgraph, 3-88
islower, 3-88
isprint, 3-88
ispunct, 3-88
isspace, 3-88
isupper, 3-88
isxdigit, 3-88
labs, 3-90
ldexp, 3-91
ldiv, 3-92
localtime, 3-93
log, 3-94
log10, 3-95
longjmp, 3-96
malloc, 3-99
memchr, 3-100
memcmp, 3-101
memcpy, 3-102
memmove, 3-103
memset, 3-104
mktime, 3-107
modf, 3-108
perror, 3-113
pow, 3-115
printf, 3-116
putc, 3-122
putchar, 3-123
puts, 3-124
rand, 3-128
realloc, 3-130
remove, 3-131
rename, 3-132
rewind, 3-133

scanf, 3-136
setbuf, 3-145
setjmp, 3-147
setvbuf, 3-148
signal, 3-150
sin, 3-153
sinh, 3-154
sprintf, 3-155
sqrt, 3-156
srand, 3-158
sscanf, 3-159
strcat, 3-163
strchr, 3-164
strcmp, 3-165
strcoll, 3-167
strcpy, 3-168
strcspn, 3-169
strlen, 3-170
strncat, 3-171
strncmp, 3-172
strncpy, 3-173
strpbrk, 3-174
strrchr, 3-175
strspn, 3-176
strstr, 3-177
strtod, 3-178
strtok, 3-179
strxfrm, 3-181
tan, 3-182
tanh, 3-183
time, 3-184
tmpfile, 3-185
tmpnam, 3-186
tolower, 3-187
toupper, 3-188
ungetc, 3-189
va_arg, 3-191
va_end, 3-191
va_start, 3-191
vfprintf, 3-192

vprintf, 3-193
vsprintf, 3-194
asctime, 3-14
asin, 3-15
assert, 3-16
assign buffer to I/O stream, 3-145, 3-148
atan, 3-17
atan2, 3-18
atexit, 3-19
atof, 3-7, 3-20
 example, 3-20
atoi, 3-21
 example, 3-21
atol, 3-22
 example, 3-22
Aztec functions
 access, 3-11
 close, 3-28
 cotan, 3-31
 creat, 3-32
 fexecl, 3-48
 fexecv, 3-48
 fileno, 3-55
 format, 3-61
 index, 3-84
 lseek, 3-97
 movmem, 3-109
 open, 3-110
 qsort, 3-125
 ran, 3-127
 read, 3-129
 rindex, 3-134
 sbrk, 3-135
 srand, 3-157
 unlink, 3-190
 write, 3-196

B

block operation functions
memchr, 3-100

memcmp, 3-101
memcpy, 3-102
memmove, 3-103
memset, 3-104
movmem, 3-109
buffer, 2-12, 2-15
 default size, 2-11
buffered I/O, 2-11, 2-14

C

c.lib, 2-2, 3-7
c32.lib, 3-7
c8.lib, 3-7
calloc, 2-20, 3-23
ceil, 3-24
change file position, 3-97
character classification functions, 3-88
 compiler option usage, 3-89
character name length, 2-2
character type function type, 3-88
character-oriented input, 2-17
clear end of file conditions in stream, 3-25
clear error conditions in a stream, 3-25
clearerr, 2-11, 2-13, 3-25
clock, 3-27
close, 2-15 - 2-16, 3-28
close a device or file, 3-28
closing files, 2-12
closing streams, 2-10, 3-42
command line arguments, 2-4, 2-8
compare memory, 3-101
compare two strings, 3-165, 3-167, 3-172
compute exponential function, 3-40
compute integer, 3-56
compute logarithm, 3-94 - 3-95
compute quotient, 3-92
compute quotient of two integers, 3-35
compute square root, 3-156
compute the difference between two times, 3-34
compute the smallest integer, 3-24

concatenate two strings, 3-163, 3-171
console I/O, 2-6, 2-17
console input
 RAW mode, 2-19
conversion function type, 3-155
conversion functions
 atof, 3-20
 atoi, 3-21
 atol, 3-22
 format, 3-61
 ftoa, 3-75
 sprintf, 3-155
 sscanf, 3-159
 strtod, 3-178
 tolower, 3-187
 toupper, 3-188
 vsprintf, 3-194
convert a calendar time, 3-83
convert a string to a double, 3-178
convert a time value to an ASCII string, 3-33
convert an ASCII string to a signed integer, 3-21
convert an ASCII string to a signed long val, 3-22
convert ASCII string to a double number, 3-20
convert character to lower case, 3-187
convert character to upper case, 3-188
convert floating point number, 3-75
convert time, 3-93, 3-107
copy a memory block, 3-109
copy block of bytes, 3-102
copy block of memory, 3-103
copy one string to another, 3-168, 3-173
cos, 3-29
cosh, 3-30
cotan, 3-31
 error codes, 3-31
creat, 2-15
create a new file, 3-32
create name for temporary file, 3-186
create temporary file, 3-185
ctime, 3-33

D

deallocate memory block, 3-66
deallocate memory space, 3-135
delete a file, 3-131
determine accessibility of file or function, 3-11
determine time intervals, 3-27
device I/O, 2-6, 2-8, 2-17
 ioctl, 3-87
devices
 opening, 2-10
devices,opening
 See opening devices
difftime, 3-34
div, 3-35
div_t type
 defined, 3-35
documentation, 1-1
dos_packet, 3-36
dynamic allocation of standard I/O buffers, 2-21
dynamic buffer allocation, 2-20 - 2-21

E

end-of-file
 testing for, 3-46
erase file, 3-190
errno, 2-11, 2-16, 2-21 - 2-22
error, 2-11, 2-21
error codes,system independent
 E2BIG, 2-23
 EACCES, 2-23
 EAGAIN, 2-23
 EBADF, 2-23
 EDOM, 2-23
 EEXIST, 2-23
 EINVAL, 2-23
 EIO, 2-23
 EMFILE, 2-23
 ENFILE, 2-23
 ENOENT, 2-23

ENOEXEC, 2-23
ENOMEM, 2-23
ENOSPC, 2-23
ENOTTY, 2-23
ERANGE, 2-23
EROFS, 2-23
EXDEV, 2-23
error flag, 2-22
error processing, 2-21 - 2-22
errors
 global integer, 2-11
execl, 3-37
execlp, 3-37
execute a goto, 3-96
execv, 3-37
execvp, 3-37
exit, 2-8, 2-10, 3-39
exiting programs, 2-8, 3-39
exp, 3-40
 error codes, 3-40

F

fabs, 3-41
fclose, 2-10, 2-12 - 2-13, 3-42
fdopen, 2-7, 2-9, 2-13, 3-43
 example, 3-44 - 3-45
 modes, 3-43 - 3-44
feof, 2-11, 2-13, 3-46
ferror, 2-11 - 2-13, 3-47
fexec functions
 parameters, 3-48
fexecl, 3-48
fexecv, 3-48
fflush, 2-13, 3-51
fgetc, 2-13, 3-52
fgetpos, 2-13, 3-53
fgets, 2-13, 3-54
file descriptor, 2-18
 defined, 2-15
file I/O, 2-8, 2-16

opening files, 2-6
random access, 2-5
sequential access, 2-5

file pointer
 defined, 2-9

fileno, 2-7, 2-13, 3-55

files, opening
 See opening files

find a character, 3-100

find character in a string, 3-84

find last occurrence of character in a string, 3-134

flags, 2-11

float math functions, 3-182

- acos, 3-13
- asin, 3-15
- atan, 3-17
- atan2, 3-18
- ceil, 3-24
- cos, 3-29
- cosh, 3-30
- cotan, 3-31
- exp, 3-40
- fabs, 3-41
- floor, 3-56
- fmod, 3-57
- frexp, 3-69
- ldexp, 3-91
- log, 3-94
- log10, 3-95
- modf, 3-108
- pow, 3-115
- ran, 3-127
- sin, 3-153
- sinh, 3-154
- sqrt, 3-156
- srand, 3-157
- tanh, 3-183

floor, 3-56

flush an I/O stream, 3-51

fmod, 3-57

fopen, 2-9, 2-13, 3-58
format, 3-61
formatted output, 3-116
fprintf, 2-13, 3-62
fputc, 2-13, 3-63
fputs, 2-13, 3-64
fread, 2-13, 3-65
free, 2-11, 2-20, 3-66
freeseg, 3-67
freopen, 2-9, 2-13
frexp, 3-69
fscanf, 2-13, 3-70
fseek, 2-5, 2-10 - 2-11, 2-14, 3-71
 example, 3-72
fsetpos, 2-14, 3-73
ftell, 2-14, 3-74
ftoa, 3-75
 example, 3-75
fwrite, 2-14, 3-76
 example, 3-76 - 3-77

G

generate floating point random numbers, 3-127
get string of characters from stream, 3-54
geta4, 3-78
getc, 2-14, 3-79
getchar, 2-13, 3-80
getenv, 3-81
gets, 2-13, 3-82
getw, 2-14
global integer, 2-11
global names, 2-2
gmtime, 3-83

H

header files, 2-10, 2-18
heap, 2-11, 2-20
hyperbolic functions
 cosh, 3-30

sinh, 3-154
tanh, 3-183

I/O

Amiga, 2-5 - 2-7
Amiga standard, 2-9, 2-14
Aztec C, 2-5 - 2-7
Aztec standard, 2-9
buffered, 2-11, 2-14
console, 2-17
device, 2-6, 2-8, 2-17
dual access calls, 2-6
file, 2-8, 2-16
random, 2-5, 2-10
sequential, 2-10
standard, 2-3, 2-12, 2-14, 2-21
testing for errors, 3-47
unbuffered, 2-3, 2-14, 2-21

I/O calls

dual access, 2-7
index, 3-84
indicate function to be called at program end, 3-19
initialize seed value used by rand, 3-158

input

 console, using RAW mode, 2-19

int_end, 3-85

int_start, 3-86

integer math functions

 abs, 3-10

 div, 3-35

 labs, 3-90

 ldiv, 3-92

 rand, 3-128

 srand, 3-158

integer sizes, 2-2

interrupt handlers, 3-85 - 3-86

ioctl, 2-6, 2-8, 2-15, 2-18, 3-87

isalnum, 3-88

isalpha, 3-88

isascii, 3-88
isatty, 2-15
iscntrl, 3-88
isdigit, 3-88
isgraph, 3-88
islower, 3-88
isprint, 3-88
ispunct, 3-88
isspace, 3-88
isupper, 3-88
isxdigit, 3-88

L

labs, 3-90
ldexp, 3-91
ldiv, 3-92
library functions
 description of return values, 3-7
 system dependent, 3-1
 system independent, 3-1
library summary, 2-1
lmalloc, 2-21
load and start program, 3-37
localtime, 3-93
log, 3-94
 error codes, 3-94
log10, 3-95
 error codes, 3-95
longjmp, 3-96
lseek, 2-9, 2-15 - 2-16, 3-97
 examples, 3-98

M

m.lib, 2-2, 3-7
m32.lib, 3-7
m8.lib, 2-2, 3-7
ma.lib, 2-2
make unique file name, 3-105
malloc, 2-11, 2-20, 3-99

- memchr, 3-100
- memcmp, 3-101
- memcpy, 3-102
- memmove, 3-103
- memory allocation functions
 - calloc, 3-23
 - free, 3-66
 - malloc, 3-99
 - realloc, 3-130
 - sbrk, 3-135
- memset, 3-104
- mf.lib, 2-2, 3-7
- miscellaneous functions
 - _abort, 3-9
 - _exit, 3-39
 - _stkchk, 3-161
 - _stkover, 3-162
 - assert, 3-16
 - atexit, 3-19
 - execl,execlp,execv,execvp, 3-37
 - exit, 3-39
 - getenv, 3-81
 - longjmp, 3-96
 - qsort, 3-125
 - setenv, 3-146
 - setjmp, 3-147
 - mktemp, 3-105
 - example, 3-105 - 3-106
 - mkttime, 3-107
 - modf, 3-108
 - movmem, 3-109

N

- non-local transfer of control, 3-147
- nonconsole drivers, 2-17

O

- open, 2-9, 2-15 - 2-16, 3-110
 - examples, 3-111 - 3-112

modes, 3-59 - 3-60, 3-110
open standard I/O window, 3-195
opening devices, 2-10, 3-43, 3-58, 3-110
 unbuffered I/O, 2-15
opening files, 2-6, 2-9 - 2-10, 3-43, 3-58, 3-110
 for unbuffered I/O, 2-15
maximum , 2-7
maximum number allowed, 2-3
output packet, 3-36
overview of Aztec C and Amiga error processing, 2-21 - 2-22

P

parse command line, 3-26
perform formatted input conversion, 3-70, 3-136, 3-159
perror, 2-14, 3-113
 error messages, 3-113 - 3-114
pmode, 3-32
pointer to an array
 example, 2-4
positioning, 2-5
pow, 3-115
preopened devices, 2-4, 2-8
print system error message, 3-113
printf, 2-13
 conversion specifiers, 3-116
 flags field, 3-117
 format string, 3-116
 linker options, 3-121
 precision field, 3-118
 size-mod field, 3-118
 type field, 3-119 - 3-120
 width field, 3-118
push character back into input stream, 3-189
putc, 2-14, 3-122
putchar, 2-13, 3-123
puterr, 2-13
puts, 2-13, 3-124
putw, 2-14

Q

qsort, 3-125
example, 3-125 - 3-126

R

ran, 3-127
rand, 3-128
random I/O, 2-5, 2-10
random number functions
 ran, 3-127
 rand, 3-128
 srand, 3-157
 srand, 3-158
RAW mode, 2-17 - 2-18
re-allocate an existing memory block, 3-130
read, 2-9, 2-15 - 2-16, 3-129
 from a device or file, 3-129
 from stream, 3-65
 string of characters from stream, 3-82
realloc, 2-20, 3-130
redirection strings, 2-5
remove, 2-14, 3-131
rename, 2-14 - 2-15
 rename a disk file, 3-132
 reopen a stream, 3-68
 reposition a stream's position indicator, 3-133
 return a pseudo-random integer, 3-128
 return absolute value of a given number, 3-41
 return absolute value of a signed long, 3-90
 return character from a stream, 3-79
 return character from standard I/O stream, 3-52
 return character from stream, 3-80
 return current file position, 3-74
 return file descriptor, 3-55
 return file information, 3-160
 return hyperbolic sine of a double value, 3-154
 return hyperbolic tangent of a double value, 3-183
 return name of next file matching pattern, 3-141
 return pointer to a string, 3-81

return remainder of double value, 3-57
return sine of a double value, 3-153
return tangent of a double value, 3-182
return the absolute value of a signed integer, 3-10
return the arc sine of a double value, 3-15
return the arc tangent of a double value, 3-17 - 3-18
return the cosine of a double value, 3-13, 3-29
return the cotangent of a double value, 3-31
return the hyperbolic cos of a double value, 3-30
return the length of a string, 3-170
return time of day, 3-184
rewind, 2-14
rindex, 3-134

S

s.lib, 2-2, 3-7
save current file position, 3-53
sbrk, 2-21, 3-135
scanf, 2-13, 3-136
 conversion characters, 3-138 - 3-140
 conversion specification, 3-137
 format string, 3-136
 matching conversion specifiers, 3-137
 matching ordinary characters, 3-137
 matching white space characters, 3-136
scdir
 example, 3-141
scr_beep, 3-142
scr_bs, 3-142
scr_cdelete, 3-142
scr_cinsert, 3-142
scr_clear, 3-142
scr_cr, 3-142
scr_curs, 3-142
scr_cursrt, 3-142
scr_cursup, 3-142
scr_eol, 3-142
scr_home, 3-142
scr_ldelete, 3-142
scr_lf, 3-142

scr_linsert, 3-142
scr_tab, 3-142
screen manipulation functions, 3-142
 descriptions, 3-143
search for first occurrence of a character, 3-164
segload, 3-144
segment load, 3-144
segment unload function, 3-67
sequential I/O, 2-5, 2-10
set a block of memory to value, 3-104
set correct file position, 3-73
set environment variables, 3-146
set random number seed for ran, 3-157
set up a4 register, 3-78
setbuf, 2-11, 2-14, 3-145
setenv, 3-146
setjmp, 3-147
sets current location within a stream, 3-71
setvbuf, 2-14, 3-148
 mode arguments, 3-148 - 3-149
sg_flags, 2-19
sgtty field, 2-19
signal
 handling, 3-150
signal function, 3-150
 example, 3-151 - 3-152
 func parameter, 3-150
 return values, 3-151
 sig parameter, 3-150
significant character name length, 2-2
sin, 2-21, 3-153
 error codes, 3-153
sinh, 3-154
 error codes, 3-154
sort an array of records in memory, 3-125
sqrt, 2-21, 3-156
sran, 3-157
strand, 3-158
sscanf, 3-159
stack check functions, 3-161 - 3-162

standard error, 2-4
standard I/O, 2-3, 2-21
standard I/O buffers
 dynamic allocation, 2-21
standard I/O functions, 2-12, 2-14
 access, 3-11
 clearerr, 3-25
 fclose, 3-42
 feof, 3-46
 ferror, 3-47
 fflush, 3-51
 fgetc, 3-52
 fgetpos, 3-53
 fgets, 3-54
 fopen, 3-58
 fprintf, 3-62
 fputc, 3-63
 fputs, 3-64
 fread, 3-65
 freopen, 3-68
 fscanf, 3-70
 fseek, 3-71
 fsetpos, 3-73
 ftell, 3-74
 fwrite, 3-76
 getc, 3-79
 getchar, 3-80
 gets, 3-82
 perror, 3-113
 printf, 3-116
 putc, 3-122
 putchar, 3-123
 puts, 3-124
 remove, 3-131
 rename, 3-132
 rewind, 3-133
 scanf, 3-136
 setbuf, 3-145
 setvbuf, 3-148
 tmpfile, 3-185

tmpnam, 3-186
ungetc, 3-189
vfprintf, 3-192
vprintf, 3-193
standard input, 2-4
standard output, 2-4
stat, 3-160
stderr, 2-10
stdin, 2-10
stdout, 2-10
strcat, 3-163
strchr, 3-164
strcmp, 3-165
strcoll, 3-167
strcpy, 3-168
strcspn, 3-169
stream
 closing, 2-10
 defined, 2-9
 I/O errors, 2-11 - 2-12
string handling functions
 index, 3-84
 rindex, 3-134
 strcat, 3-163
 strchr, 3-164
 strcmp, 3-165
 strcoll, 3-167
 strcpy, 3-168
 strcspn, 3-169
 strlen, 3-170
 strncat, 3-171
 strncmp, 3-172
 strncpy, 3-173
 strpbrk, 3-174
 strrchr, 3-175
 strspn, 3-176
 strstr, 3-177
 strtok, 3-179
 strxfrm, 3-181
string searching, 3-169, 3-174 - 3-177, 3-181

strings

- compare two strings, 3-165, 3-167, 3-172
 - concatenate two strings, 3-163, 3-171
 - convert to a double, 3-178
 - copy one string to another, 3-168, 3-173
 - return the length of a string, 3-170
 - search for first occurrence of character, 3-164
 - string searching, 3-169
 - tokenize, 3-179
- strlen**, 3-170
strncat, 3-171
strcmp, 3-172
strcpy, 3-173
strpbrk, 3-174
 strrchr, 3-175
strspn, 3-176
strstr, 3-177
strtod, 3-178
strtok, 3-179
strxfrm, 3-181
- system independent error codes**
See **error codes, system independent**

T

- tan**, 3-182
- tanh**, 3-183
- terminate calling program**, 3-39
- test for an error**, 3-47
- test for EOF condition**, 3-46
- time**, 3-184
- time functions**
 - asctime**, 3-14
 - clock**, 3-27
 - ctime**, 3-33
 - difftime**, 3-34
 - gmtime**, 3-83
 - localtime**, 3-93
 - mktime**, 3-107
 - time**, 3-184
- tmpfile**, 2-14, 3-185

tmpnam, 2-14, 3-186
tokenize a string, 3-179
tolower, 3-187
toupper, 3-188
translates a time value into an ascii string, 3-14

U

unbuffered I/O, 2-3, 2-21
unbuffered I/O to nonconsole drivers, 2-17
unbuffered I/O to the Console, 2-17
unbuild real numbers, 3-69, 3-91, 3-108
ungetc, 2-14, 3-189
UNIX, 2-14, 2-21
UNIX I/O functions
 close, 3-28
 creat, 3-32
 fdopen, 3-43
 fileno, 3-55
 ioctl, 3-87
 lseek, 3-97
 mktemp, 3-105
 open, 3-110
 read, 3-129
 stat, 3-160
 unlink, 3-190
 write, 3-196
unlink, 2-15, 3-190
using I/O
 examples, 2-19

V

va_arg, 3-191
va_end, 3-191
va_start, 3-191
variable argument access, 3-191
variable argument functions
 va_arg, 3-191
 va_end, 3-191
 va_start, 3-191

verify program assertion, 3-16

vfprintf, 2-14, 3-192

vprintf, 2-13, 3-193

vsprintf, 3-194

W

write, 2-9, 2-15 - 2-16, 3-196

write a character to an I/O stream, 3-122

write a character to I/O stream, 3-63

write a character to the stdout stream, 3-123

write a string to stdout, 3-124

write formatted ASCII data, 3-192 - 3-194

write formatted data, 3-61 - 3-62

write formatted data to a buffer, 3-155

write string to I/O stream, 3-64

write to a file or device, 3-196

write to a specified stream, 3-76

writing system-independent programs, 2-18

Aztec C UniTools for the Amiga

**Version 5.0
October 1989**

Copyright 1988, 1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

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 C68k 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

This software package and document are copyrighted ©1988, 1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

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 C68k, Manx AS, Manx LN, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh 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

October 1989 Second Edition

December 1988 First Edition

Table of Contents

Chapter 1 - Overview

Introduction	1-1
About This Manual	1-2
Technical Support	1-3

Chapter 2 - Diff

The Conversion List	2-1
Conversion Items	2-2
The Command Line	2-2
The Affected Lines	2-3
The -b Option	2-4
Differences Between the UNIX and Manx Versions of diff	2-4

Chapter 3 - Grep

Description of grep	3-1
OPTIONS	3-3
INPUT FILES	3-3
PATTERNS	3-4
Differences Between the Manx and UNIX Versions of grep	3-7
OPTION DIFFERENCES	3-7
PATTERN DIFFERENCES	3-7
Examples	3-7
SIMPLE STRING MATCHING	3-7
MORE EXAMPLES	3-8

Chapter 4 - Make

Preview	4 - 2
Note to Previous Users	4 - 2
The Basics	4 - 2
What make Does	4 - 3
The Makefile	4 - 3
RULES	4 - 6
make's Use of Rules	4 - 6
Example	4 - 7
Interaction of Rules and Dependency Entries	4 - 8
Advanced Features	4 - 8
Dependent Files	4 - 8
MACROS	4 - 9
Using Macros	4 - 10
Defining Macros in a makefile	4 - 10
Defining Macros in a Command Line	4 - 11
Macros Used by Built-In Rules	4 - 11
Special macros	4 - 11
RULES	4 - 12
Rule Definition	4 - 12
Built-in Rules	4 - 14
COMMANDS	4 - 14
Allowed Commands	4 - 14
Logging Commands and Aborting make	4 - 15
Long Command Lines	4 - 15
Makefile Syntax	4 - 15
Comments	4 - 16
Line Continuation	4 - 16
Starting make	4 - 17
The Command Line	4 - 18
make's Standard Output	4 - 18

Differences between the Manx and UNIX make Programs	4 - 19
Examples	4 - 20
The makefile in the 'libc' Directory	4 - 21
Makefile for the 'sys' Directory	4 - 22
Makefile for the 'misc' Directory	4 - 23

Chapter 5 - Z Editor

Requirements	5 - 2
Components	5 - 3
Getting Started	5 - 3
CREATING A NEW PROGRAM	5 - 3
The Screen	5 - 4
Modes of Z	5 - 4
Insert Mode	5 - 4
Exiting Z	5 - 5
EDITING AN EXISTING FILE	5 - 5
Starting and Stopping Z	5 - 6
Command Line Options	5 - 6
The Cursor	5 - 7
Moving Around in the Text: Scrolling	5 - 7
Moving Around in the Text: the Go Command	5 - 8
Moving Around in the Text: String Searching	5 - 8
Finely Tuned Moves	5 - 9
Deleting Text	5 - 9
More Insert Commands	5 - 10
Summary	5 - 10
More Commands	5 - 11
THE SCREEN	5 - 11
Displaying Unprintable Characters	5 - 11
Displaying Lines That Do Not Fit on the Screen	5 - 11
Commands	5 - 12

Special Keys	5 - 12
PAGING AND SCROLLING	5 - 12
SEARCHING FOR STRINGS	5 - 13
Additional String-Searching Commands	5 - 13
Regular Expressions	5 - 14
Enabling Extended Pattern Matching	5 - 15
LOCAL MOVES	5 - 16
Moving Around on the Screen:	5 - 16
Moving within a Line	5 - 16
Word Movements	5 - 17
Moves within C Programs	5 - 18
Marking and Returning	5 - 19
Adjusting the Screen	5 - 20
MAKING CHANGES	5 - 20
Small Changes	5 - 20
Operators for Deleting and Changing Text	5 - 21
Deleting and Changing Lines	5 - 22
Moving Blocks of Text	5 - 23
Duplicating Blocks of Text: the Yank Operator	5 - 23
Named Buffers	5 - 24
Moving Text between Files	5 - 25
Shifting Text	5 - 26
Undoing and Redoing Changes	5 - 26
INSERTING TEXT	5 - 26
Additional Insert Commands	5 - 27
Insert Mode Commands	5 - 27
Autoindent	5 - 28
MACROS	5 - 28
Immediate Macro Definition	5 - 28
Examples	5 - 29
Indirect Macro Definition	5 - 31
Reexecuting Macros	5 - 31

Wrapping Around During Macro Execution	5 - 32
EX-LIKE COMMANDS	5 - 32
Addresses in Ex Commands	5 - 33
The Substitute Command	5 - 34
The c Option	5 - 34
The g Option	5 - 34
Examples	5 - 35
The "&" (Repeat Last Substitute) Command	5 - 35
QUIKFIX COMMANDS	5 - 36
THE OPTION FILE	5 - 36
The ZOPT Environment Variable	5 - 37
Setting Options for a File	5 - 37
STARTING AND STOPPING Z	5 - 37
Starting Z	5 - 38
Starting Z without a Filename	5 - 38
Starting Z with a List of Files	5 - 38
Stopping Z	5 - 39
ACCESSING FILES	5 - 39
Filenames	5 - 40
Writing Files	5 - 40
Reading Files	5 - 41
Editing Another File	5 - 41
File Lists	5 - 43
Tags	5 - 44
The ctags Utility	5 - 45
OPTIONS	5 - 45
DIFFERENCES BETWEEN Z AND VI	5 - 47
Command Summary	5 - 49
Starting Z	5 - 49
The Display	5 - 49
Options	5 - 49

Adjusting the Screen	5 - 49
Positioning within File	5 - 50
Marking and Returning	5 - 50
Line Positioning	5 - 50
Character Positioning	5 - 51
Words and Paragraphs	5 - 51
Insert and Replace	5 - 51
Corrections During Insert	5 - 52
Operators	5 - 52
Miscellaneous Operations	5 - 52
Yank and Put	5 - 52
Undo and Redo	5 - 53
Macros	5 - 53
QuikFix Commands	5 - 53
Colon Commands	5 - 53

OVERVIEW

DIFF

GREP

MAKE

Z EDITOR

OVERVIEW

1

Chapter 1 - Overview

Introduction

Welcome to UniTools--the perfect supplement to your Aztec C software development system! The utilities in UniTools work with Aztec C to give you tremendous programming power and flexibility.

If you have purchased the **Developer Level** of Aztec C for the Amiga, then UniTool utilities and SDB have been included on your Aztec C disks.

If, however, you have purchased the **Professional Level** of Aztec C for the Amiga, it does not include the **make**, **grep**, and **diff** portions of Unitools, even though the documentation for all of these utilities has been included here. However, please note: The **Professional Level** can be easily upgraded to the **Developer Level**. See page 4-4 of your *Aztec C User Guide* for information on how to contact the Update Department.

UniTools includes the utilities **diff**, **grep**, **make**, and **Z editor**. Each of these important utilities is designed to greatly enhance your programming efficiency, as follows:

- The **diff** utility finds the areas of difference in your source files and tells you the changes that must be made to make the files identical.

- The **grep** utility allows you to match patterns in your program and print the locations. This makes it easier for you to either search for, or reference, specific strings and function calls.
- The **make** utility allows you to build large applications without the time-consuming process of using batch files that contain several compile and link command lines. This saves you from having to type command statements over and over.
- The Aztec C **Z editor** is a powerful text editor that allows you to create and edit C source programs. The **Z editor** is similar to the UNIX vi editor.

About This Manual

This manual describes the utilities, **diff**, **grep**, **make**, and **Z**, and gives the commands used in each. It also includes an index of keywords and subjects to make the information in the manual more accessible.

Throughout this manual, we use the following conventions:

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 likely will include explanation or definition of term
{choice1 choice2}	braces and a vertical bar mean that you have a choice between two or more items
<i>placeholders</i>	information that must be supplied by the user; for example, <i>filename</i> , <i>range</i> , <i>identifier</i> ,etc.
output	to show text that is generated by the computer
<i>[optional]</i>	to show optional information

Technical Support

At Manx, we build dependability into our products. However, if you should find that you need help, rest assured that we provide it.

Refer to your Aztec C Documentation for information on the technical support that Manx provides. If you have any problems with UniTools, follow the procedures outlined in the **Technical Support** chapter so that we may best be able to assist you.

DIFF

2

Chapter 2 - Diff

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.

NAME

`diff` - Source file comparison utility

SYNOPSIS

`diff [-b] file1 file2`

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`.

DIFF

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 following command line says to add line **3** of **file2** after line **5** of **file1**:

5a3

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

16a8,10

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

100,150d75

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

32c33

and the next command says to replace lines **453 through 500** in **file1** with lines **490 through 499** in **file2**:

453,500c490,499

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 of diff

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

3

Chapter 3 - Grep

Description of grep

grep is a program, similar to the UNIX **grep**, that searches for a designated pattern within a specified group of files. The pattern can consist of characters, strings, or a group of strings.

NAME

grep - pattern matching program

SYNOPSIS

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

For example, you could use **grep** to find the function **hello**, as follows:

```
grep hello *.c
```

This command would print out all of the files which reference **hello**.

grep will also allow you to search for patterns in particular places within a file, such as at the beginning of lines or at the ends of lines.

grep will allow you to search multiple files for the pattern you specify. **grep** usually takes two arguments: *pattern*, which indicates what you are looking for, and *file1, file2,etc.* which defines the files you want to search. *file1, file2, etc.* can be the name(s) of a specific file, or can contain wildcard characters. For example, you can say:

```
grep hello *.c
```

or

```
grep hello mike.c
```

or

```
grep hello mike.??1
```

The pattern choices cover a wide range, from simple to complex. For example, you could specify as a *pattern* the letter **m**

```
grep m *.h
```

the function name **hello()**

```
grep hello() *.c
```

or even "the name **hello** but only when it appears at the beginning of a line and with nothing else on that line"

```
grep ^hello$ *.c
```

where **^** means must start at beginning of the line, and **\$** means it is the last item on the line. (See Figure 3.1.)

Typing **grep** on a line all by itself will cause it to search the standard input. Thus, **grep** will return anything you type because it considers your keyboard its source of input.

OPTIONS

The following options are supported by grep:

- v Print all lines that do not match the pattern.
- c Print just the name of each file and the number of matching lines that it contained.
- 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.
- f A character in the pattern will match both its upper- and lowercase equivalent.

INPUT FILES

The files parameter is a list of files to be searched. 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

See Figure 3.1 for a brief synopsis of the special patterns you can use with **grep**.

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.

Special Patterns Used With grep

- . matches anything except carriage return or new line
- [] matches any one character within the brackets
- [^] matches all characters except the ones within the brackets
- ^ matches a pattern only if it is the first item on the line
- \$ matches a pattern only if it is the last item on the line.
- x* matches 0 or more occurrences of the character x

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

Figure 3.1



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.

► []

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
```

► \$ and ^

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
```

► *

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`.

► \

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`.

Differences Between the Manx and UNIX Versions of grep

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.

Examples

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.

MORE EXAMPLES

If you wish to find all lines in a group of files with a **.c** extension which contain calls to **sprintf** and **fprintf**, but not **printf**, the command

```
grep -n [fs]printf *.c
```

will work.

If, on the other hand, you wish to find only **printf** calls in your **.c** files, and not **sprintf** or **fprint**, then you can use this **grep** command:

```
grep -n [^a-z]printf *.c
```

Suppose you need to see all calls to the function **Lex**, but only within **if** statements. You would enter

```
grep if.*Lex *.c
```

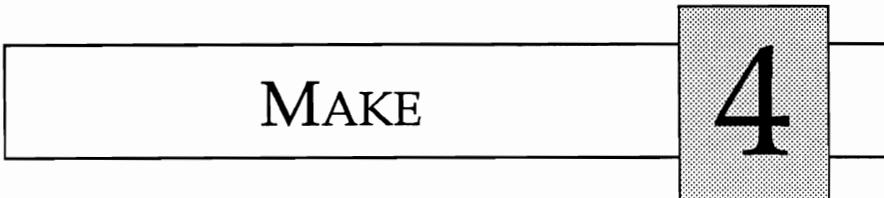
Finally,

```
grep ^[a-z,A-Z,_][^;)]*)$ *.c
```

will find all function definitions in your **.c** files, assuming that all definitions take place on one line and that a comment is not on the same line as the definition. This is how it works:

1. The **^** states that the following pattern must be the first item on the line.
2. **[a-z, A-Z, _]** states that the first character can only be a letter or a **_**. Preprocessor directive, like **#include** and similar statements, will not fit the pattern.

3. The pattern `[^;)]*` says "do not match until you reach a ; or a)."
4. The `)$` pattern will only match a) at the end of the line. If the previous line matched anything other than a), which could only be an end of line or a ;, then the whole pattern will fail. In this way, only function definitions which end in a) will match.



Chapter 4 - Make

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. The **make** program discussed in this chapter is the Manx **make**.

NAME
make - Program maintenance utility
SYNOPSIS
make [*options*] [*name1 name2 ...*]

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

Note to Previous Users

One notable change has been implemented in this release of **make**: Default rules, when applied to files in other directories, will place the results in those other directories. Prior to Version 5.0, the results were placed in the current directory.

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.o sub1.o sub2.o
      ln -o prog prog.o sub1.o sub2.o -lc
```

This entry says that the file **prog** depends on the files **prog.o**, **sub1.o**, and **sub2.o**. It also says that if **prog** is not current, make should execute the **ln** command. make considers **prog** to be current if it exists and if it has been modified more recently than **prog.o**, **sub1.o**, and **sub2.o**.

The above entry describes only the dependence of **prog** on **prog.o**, **sub1.o**, and **sub2.o**. It does not define the files on which the **.o** files depend. For that, we need either additional dependency entries in the makefile or a rule that can be applied to create **.o** files from **.c** files.

For now, we will add dependency entries in the makefile for **prog.o**, **sub1.o**, and **sub2.o**, 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 modify the makefile to make use of make's built-in rule for creating a **.o** file from a **.c** file.

Suppose that the **.o** 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 long-winded makefile could be used to explicitly define all the information needed to create **prog**

```
prog: prog.o sub1.o sub2.o
      ln -o prog prog.o sub1.o sub2.o -lc
prog.o: prog.c
      cc prog.c
sub1.o: sub1.c defs.h
      cc sub1.c
sub2.o: sub2.c defs.h
      cc sub2.c
```

This makefile contains four dependency entries: for **prog**, **prog.o**, **sub1.o**, and **sub2.o**. 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.o** current, you could enter

```
make sub2.o
```

A makefile can contain dependency entries for unrelated files. For example, the following dependency entries can be added to the above makefile:

```
hello: hello.o
    ln hello.o -lc
hello.o: hello.c
    cc hello.c
```

With these dependency entries, you can tell **make** to make **hello** and **hello.o**, in addition to **prog** and its object files.

RULES

You can see that the makefile describing a program built from many .o files would be huge if it had to explicitly state that each .o 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 how you write makefiles that make use of **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.

One rule built into **make**, for converting .c files into .o files, says

"a file having extension .o depends on the file having the same basic name, with extension .c. To make current such a .o file, execute the command

cc x.c

where *x* is the name of the file."

Another built-in rule exists for converting .asm files into .o files, using the Manx assembler.

Example

The .c to .o rule allows us to abbreviate the long-winded makefile shown above as follows:

```
prog: prog.o sub1.o sub2.o
      ln -o prog prog.o sub1.o sub2.o -lc
      sub1.o sub2.o: defs.h
```

In this abbreviated makefile, a dependency entry for **prog.o** is not needed; using the built-in .c-to-.o rule, **make** infers that the **prog.o** depends on **prog.c** and that the command **cc prog.c** will make **prog.o** current.

The abbreviated makefile says that both **sub1.o** and **sub2.o** depend on **defs.h**. It does not say that they also depend on **sub1.c** and **sub2.c**, respectively, or that the compiler must be run to make them current; **make** infers this information from the .c to .o rule. The only information given in the dependency entry is that which **make** could not infer by itself: that the two object files depend on **defs.h**.

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.o** entry in the long-winded makefile shown earlier in this section was not needed, because its information could be inferred by the **.c** to **.o** rule. And the dependence of **sub1.o** and **sub2.o** 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 **.c** to **.o** 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:  
    cc -DFLOAT add.c
```

causes **add.o** to be compiled using the specified command rather than the command specified by the **.c** to **.o** rule. **make** still infers the dependence of **add.o** on **add.c**, using the **.c** to **.o** rule, however.

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

make supports dependencies on files in directories other than the current directory. This is best illustrated by the following example:

Suppose the current directory is **/src1** and that the program is being built from source there and in the **/src2** directory. The makefile line would look like this:

```
program: /src/main.o /src2/sub.o  
        ln -o program /src1/main.o /src2/sub.o -lc
```

This builds **main.o** and **sub.o** from their sources, putting the object files (the **.o**'s) in the same directory as the sources. The **.o** file will always be placed in the same directory as the source file.

Note: Unfortunately, there is no way to say "all my .o's are in one directory and all my .c's are in another."

Any further references to these files should be with the same path or else **make** will be confused.

For object files to be dependent on files in another directory, the full pathname must be used as in:

```
/src2/sub.o : /header/defs.h
```

There are a few conventions that you must be aware of when naming your files. These are as follows:

- 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.o b.o c.o d.o**. Then the dependency entries:

```
prog: prog.o a.o b.o c.o d.o  
    ln prog.o a.o b.o c.o d.o  
a.o b.o c.o d.o: defs.h
```

within a makefile could be abbreviated as:

```
prog: prog.o $(OBJECTS)  
    ln prog.o $(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.o b.o c.o d.o
```

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. Be careful about your macro definitions if you are using **make** on systems other than the Amiga. Some operating systems do not support quotes around text, so any white space in the macro text causes the macro definition to terminate prematurely.

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.o y.o z.o:
  cc -DFLOAT $*.c
```

When **make** decides that **x.o** needs to be recreated from **x.c**, it will assign **\$*** to the string **x**, and the command

```
cc -DFLOAT x.c
```

will be executed. Similarly, when **y.o** or **z.o** is made, the command

```
cc -DFLOAT y.c
```

or

```
cc -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

Earlier we presented the basic features of rules: what they are and how they are used. We noted that rules could be defined in a makefile and that some rules are built into **make**. In the following paragraphs, we describe how rules are defined in a makefile and list the built-in rules.

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:  
cc -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

```
cc -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. For example, the built-in rule for creating a .o file from a .c file looks like this:

```
.c.o:  
cc $(CFLAGS) -o $@ $*.c
```

If you want the rule to generate an assembly language listing, include the following rule in your makefile:

```
.c.o:  
cc $(CFLAGS) -a $*.c  
as -ZAP -l -o $@ $*.asm
```

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 **.c** to **.o** rule, you can include the line

```
CFLAGS = -T -DDEBUG
```

in the makefile. Another way to accomplish the same result is to redefine the **.c** to **.o** rule in the makefile; this, however, would use more lines in the makefile than the macro redefinition.

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.o b.o c.o d.o  
      ln -o prog a.o b.o c.o d.o -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 cc, as, or ln, or that start a program created with cc, as, ln, and c.lib, 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)  
      ln -o prog $(OBJECTS) -lc
```

This will result in a very long command line being passed to **ln**.

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  
# an 8080 object module  
# from a C source file:  
#  
.c.o80:  
    cc80 -o cc.tmp $*.c  
    as80 -ZAP -o $*.o80 cc.tmp
```

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.o fprintf.o format.o\  
      scanf.o fscanf.o scan.o\  
      getchar.o getc.o
```

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.o printer.o \
    in.o \33
    out.o
lib -o driver.lib driver.o
    printer.o \
    in.o out.o
```

This second example could have been more cleanly expressed using a macro:

```
DRIVOBJ= driver.o printer.o\
    in.o out.o
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

This example shows a makefile for making several programs. Note the entry for **arc**. This does not result in the generation of a file called **arc**; it is only used so that **arcv** and **mkarcv** can be generated by entering **make arc**.

```
#  
# rules:  
#  
.c.o80:  
    cc80 -DTINY -o $@ $*.c  
#  
# macros:  
#  
OBJ=make.o parse.o scandir.o \  
dumptree.o rules.o command.o  
#  
# dependency entry for making make:  
make: $(OBJ)  
    ln -o make $(OBJ) -lc  
# dependency entries for making arcv & mkarcv:  
#  
arc: mkarcv arcv  
mkarcv: mkarcv.o  
    ln -o mkarcv mkarcv.o -lc  
arcv: arcv.o  
    ln -o arcv arcv.o -lc  
#  
# dependency entries for making  
# CP/M-80 versions of arcv & mkarcv:  
#  
mkarcv80.com: mkarcv.o80  
    ln80 -o mkarcv80.com mkarcv.o80 -lt -lc  
arcv80.com: arcv.o80  
    ln80 -o arcv80.com arcv.o80 -lt -lc  
$(OBJ): libc.h makefile
```

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
        rm my.lib
        libutil -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 ..
```

Makefile for the 'sys' Directory

```
REL=asctime.o bdos.o begin.o chmod.o \  
     croot.o csread.o ctime.o \  
     dostime.o dup.o exec.o execl.o execlp.o \  
     execv.o execvp.o \  
     fexec.o fexeccl.o fexecv.o fftime.o \  
     getcwd.o getenv.o \  
     isatty.o localtim.o mkdir.o open.o stat.o \  
     system.o time.o\  
     utime.o wait.o dioctl.o ttyio.o access.o  
     syserr.o \  
 
```

```
COPT=
HEADER=../header
.c.o:
cc $(COPT) -I$(HEADER) $*.c -o $@
sqz $@
.asm.o:
as $*.asm -o$@
sqz $@
all: $(REL)
clean:
rm *.o
```

Makefile for the 'misc' Directory

```
REL= atoi.o atol.o calloc.o ctype.o format.o \
      malloc.o \
      qsort.o sprintf.o sscanf.o fformat.o fscanf.o
COPT=
HEADER=../header
.c.o:
cc $(COPT) -I$(HEADER) $*.c -o $@
sqz $@
.asm.o:
as $*.asm -o $@
sqz $@
all: $(REL)
fformat.o: format.c
cc -I$(HEADER) -DFLOAT format.c -o fformat.o
fscanf.o: scan.c
cc -I$(HEADER) -DFLOAT scan.c -o fscanf.o
clean:
rm *.o
```




Chapter 5 - 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.
- An interface to the Manx QuikFix facility. QuikFix launches the editor directly from the compiler, loads the source file, positions to the first statement in error, and displays the error message. You can then correct the error and automatically go to the next error. After you correct the errors, you enter, :wq, the changes are saved, and the compiler is re-launched. If necessary, the process repeats. QuikFix cuts out administrative time and eliminates the bother of matching up error messages and source lines. There is no fumbling around, there are no annoying distractions. QuikFix orchestrates the motions and lets you focus on the solutions.
- 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 to exit 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! QUILTS Z without writing anything to the file being edited.

Command Line Options

With Aztec C version 5, z supports the two following options to be used in conjunction with QuikFix.

- e The option -e when z is invoked causes z to parse the AztecC.Err file generated by the compiler. z reads the first line of the file and determines the name of the file in which the errors occurred. It then reads and stores all the errors associated with the file up to 25 errors. Additional errors for this file are left in the AztecC.Err file for later parsing. The lines read are removed from the file. If all are read, the file is removed. z then reads the target

source file and positions the cursor on the correct line and column and displays the appropriate error or warning message.

- o The option **-o** causes **z** when run in the background to open a window at the specified position with the specified size and title. This string is appended to the "RAW:" which is used to open the window and follows the standard Amiga conventions for console windows.

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 ^AD rather than Control-D, for brevity. Thus, the scroll up and scroll down commands are represented as ^AU and ^AD, 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 non-whitespace character on the line.
CR (carriage return)	Moves the cursor down one line, to the first non-whitespace 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

```
mx
```

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 accidentally 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 motion 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:

```
:>-sx
```

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:

```
:>+wdwi and 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.
- i** and **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:

:[ranges] /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.

QUIKFIX COMMANDS

With version 5 of Aztec C, **z** now supports the following five commands to be used in conjunction with the QuikFix facility.

- :cc redisplay current error
- :cp display previous error
- :cn display next error

Using any of the above commands will load the source file if not already loaded and move to the appropriate line and column and display the error message.

- :cf This command is used to parse the next set of errors from the **AztecC.Err** file. Saying **z -e** is the same as saying **z** and then giving the :cf command. This can be used if more than 25 errors are in the error file or if the error file contains errors for more than one source file.
- :cq This command works the same as the :q command to exit **z**, except that it causes **z** to return an error code instead of the normal non-error code. This is used to cause the compiler not to recompile the module when **z** terminates

For additional information on the QuikFix facility, refer to the **Getting Started** Chapter of the *Aztec C User Manual*

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`

The ZOPT Environment Variable

The ZOPT environment variable defines the file in which :set options are found.

For example, the following command says that the options are in the file `zopt.cmd`:

`set ZOPT=zopt.cmd`

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 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.

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.

Three 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.
- :cq** Works the same as the :q command to exit the editor, except that it causes the editor to return an error code instead of the normal non-error code. This is used to tell the compiler not to recompile the module when the editor terminates.

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 <i>filename</i>	Edit the specified file.
:e! <i>filename</i>	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 option 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 **crtdrv.c**, you may come across a call to the function **pcdrv**, and may want to take a look at it. By positioning the cursor at the beginning of the word **pcdrv** 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.

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.
- cs** Clear Screen Option. When this option is enabled, the repaint command (Control-L) will first clear the entire screen and then repaint it. If it is disabled, each line of text on the screen will be redrawn and then cleared to the end of the line. Some systems redraw the screen much faster if this option is enabled.
- 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 pre-configured 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.

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} indicates whether or not macros perform their operations silently {on | off}

cs={1|0} determines whether screen is cleared during repaint {on | off}

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 <i>pat</i> searching forwards
?pat	move cursor to <i>pat</i> 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
mx	mark position with letter <i>x</i>
'x	to mark <i>x</i>
'x	first nonwhite at mark <i>x</i>

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
fx	find character <i>x</i> forward
Fx	find character <i>x</i> backwards
tx	position before character <i>x</i> forward
Tx	position before character <i>x</i> backwards
;	repeat last f, F, t or T
,	repeat last f, F, t or T in reverse direction
l	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 <i>x</i>
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
"xp	put from buffer <i>x</i>
"xy	yank to buffer <i>x</i>
"xd	delete to buffer <i>x</i>

Undo and Redo

u	undo last change
U	restore current line
.	repeat last change command

Macros

@X	execute macro in buffer x
"xv	execute macro in buffer x
@@	repeat last macro
v	repeat last macro

QuickFix Commands

:cc	redisplay current error
:cn	display next error
:cp	display previous error
:cq	terminate with error condition
:cf	parse next set of errors

Colon Commands

:e name	edit file <i>name</i>
:e	redit last file
:el name	edit file <i>name</i> , discarding changes
:el!	redit last file, discarding changes
:e #	edit alternate file
^^	edit alternate file
:el! #	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 <i>name</i> into current file
:w	write back to file being edited
:wq	write back to file and quit
:w name	write to file <i>name</i> if does not exist
:w! name	write to file <i>name</i> , 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 <i>name</i>	change <i>name</i> of current file
^G	show current file and line
:n	edit next file in list
:n!	edit next file in list, discarding change
:n <i>arg1 arg2....</i>	specify new list
:rew	point back to beginning of list
:rew!	point back to beginning, discarding changes
:ta <i>tag</i>	position to <i>tag</i> in appropriate file, searches file pointed to by environment variable TAGS if tags does not exist or <i>tag</i> is not found in it.
^]	same as :ta using word at cursor
:ta! <i>tag</i>	position to <i>tag</i> , discarding changes
:>macro	specify and execute immediate macro
:set <i>opt1=val opt2=val ...</i>	set editor options
:se <i>opt1=val opt2=val ...</i>	set editor options
:set all	display current option settings
:[range]s/pat/rep/[options]	substitute <i>rep</i> format in <i>range</i>
:[range]&	repeat last substitute command

Index

\$,grep, 3-2
*,grep, 3-2 - 3-3
?,grep, 3-3

A

aborting make, 4-15
accessing files, Z editor, 5-39, 5-45
adding lines, diff, 2-2
AFLAGS macro, make, 4-11, 4-14
 value, 4-14
arcv, make, 4-20
arguments, grep, 3-2
assembly language source file, make, 4-21
autoindent, Z editor, 5-5, 5-28

B

backslash, make, 4-16
backspace key, Z editor, 5-5
backup files, Z editor, 5-6
batch commands, make, 4-14
blanks, diff, 2-4
buffer, Z editor, 5-7, 5-23 - 5-24
built-in rules, make, 4-14

C

CFLAGS macro, make, 4-11, 4-14
 value, 4-14
changing lines, diff, 2-2
character positioning commands, Z editor, 5-51
character strings
 make, 4-2, 4-9
 Z editor, 5-14
colon commands, Z editor, 5-53
command line, diff, 2-2

command line, make, 4-15, 4-18
 makefile, 4-10
 maximum length, 4-16
 optional parameters, 4-18
command mode, Z editor, 5-6
command sequence, make, 4-2
components, Z editor, 5-3
control keys, Z editor, 5-12
conversion lists, diff, 2-1
 types of operations, 2-2
correction insert commands, Z editor, 5-52
creating new program, Z editor, 5-3
ctags utility, Z editor, 5-3, 5-45
current directory, make, 4-17
cursor motion commands, Z editor, 5-17
 examples, 5-9
cursor positioning, Z editor, 5-7

D

default drive, make, 4-17
definition
 diff, 1-1
 grep, 1-2
 make, 1-2
 makefile, 4-3
 named buffers, Z editor, 5-24
 regular expression, Z editor, 5-14
 Z editor, 1-2, 5-1
deleting lines, diff, 2-2
deleting text, Z editor, 5-21 - 5-23
dependency entries, make
 definition, 4-3
 makefile, 4-3, 4-10, 4-18
dependency lines, make
 maximum length, 4-17
dependent files, make, 4-8 - 4-9
Developer Level, 1-1
diff
 examples, 2-2 - 2-4
 -b option, 2-4

adding lines, 2-2
blanks, 2-4
changing lines, 2-2
command line, 2-2
conversion items, 2-2 - 2-3
conversion list, 2-1
definition, 1-1
deleting lines, 2-2
directory names, 2-5
file length, 2-4
range of lines, 2-2
replacing lines, 2-2
UNIX options not supported, 2-4 - 2-5
directory names, diff, 2-5
directory, make
 recording time and date, 4-3
display commands, Z editor, 5-49
duplicating text, Z editor, 5-23 - 5-24

E

editing, Z editor, 5-5, 5-41 - 5-42
 examples, 5-42
 another file, 5-41
 syntax, 5-41
environment variable
 ZOPT, 5-36
escape key, Z editor, 5-12
Ex-like commands, Z editor
 "&" command, 5-35
 addresses, 5-33
 arguments, 5-32
 c option, 5-34
 g option, 5-34
 repeat last substitute command, 5-32
 substitute command, 5-32, 5-34

F

file extension, make, 4-6
file length, diff, 2-4

file list feature, Z editor, 5-43
file parameter, grep, 3-3
file specification, grep, 3-2

G

go command, Z editor, 5-8
grep, 5-14
 definition, 1-2, 3-2 - 3-3, 3-7 - 3-8
 \$, 3-5
 -c option, 3-3
 -f option, 3-3
 -l option, 3-3
 -n option, 3-3
 -v option, 3-3
 [[^]], 3-4
 [], 3-5
 \, 3-6
 ^, 3-5
 arguments, 3-2
 file parameter, 3-3
 file specification, 3-2
 multiple file support, 3-2
 multiple options, 3-8
 patterns, 3-1 - 3-2
 simple string matching, 3-7
 special patterns examples, 3-5 - 3-6, 3-8
 special patterns list, 3-4
 standard input, 3-3
 UNIX similarities and differences, 3-1, 3-7
 wildcard characters, 3-2 - 3-3
 x*, 3-6

indirect macro definition, 5-31
insert mode, Z editor, 5-10
 ^W command, 5-5
 command list, 5-26
 commands, 5-27, 5-51
 exiting, 5-5, 5-12

i, 5-4
memory-resident buffer, 5-4
interfile dependencies
makefile, 4-1

L

line movement commands, Z editor, 5-15 - 5-17
local moves, Z editor, 5-16
logging commands, make, 4-15

M

macro buffer, Z editor, 5-28
macros, make, 4-9
 examples, 4-10 - 4-12
 \$\$, 4-11
 \$, 4-11
 \$@, 4-11
 AFLAGS, 4-11, 4-14
 built-in rules, 4-11
 capabilities, 4-2
 CFLAGS, 4-11, 4-14
 command line, 4-12
 defining in command line, 4-11
 invoking, 4-10
 names, 4-2
 naming, 4-10
 option -DFLOAT, 4-12
 used by built-in rules, 4-11
macros, Z editor, 5-2, 5-29 - 5-32, 5-53
 immediate macro definition, 5-28 - 5-30
 indirect macro definition, 5-31
 reexecuting, 5-31
make
 examples, 4-21
 aborting, 4-15
 advanced features, 4-8
 arcv, 4-20
 assembly language listing, 4-13
 assembly language source file, 4-21

backslash, 4-16
batch commands, 4-14
built-in rules, 4-11, 4-14
C source file, 4-21
character strings, 4-2, 4-9
colon, 4-13
command execution, 4-15
command line, 4-12, 4-15, 4-18
command line length, 4-15 - 4-16
command line parameters, 4-18
command sequence, 4-2
comments, 4-16
creating a makefile, 4-2
current directory, 4-17
default drive, 4-17
default rules, 4-2
definition, 1-2
dependency lines, length, 4-17
dependent files, 4-8 - 4-9
disk file, 4-14
file extension, 4-6
filenames, 4-9
interfile dependencies, 4-1
line continuation, 4-16
logging commands, 4-15
macro definition example, 4-16
makefile, 4-1
mkarcv, 4-20
naming conventions, 4-9
null character string, 4-11, 4-14
object files, 4-9
object files, removing, 4-21
operating system commands, 4-14
overriding rules, 4-8
parameters, 4-18
path, 4-9
prerequisite files, 4-3 - 4-4
printing error messages, 4-18
recording time and date, 4-3
redirecting standard output, 4-18

return code, 4-15
rule definition, 4-12
rule-processing capability, 4-2
rules, 4-2, 4-6, 4-12
rules built-in, 4-6
rules, target extension, 4-6
sequence of commands, rules, 4-6
source extension, 4-12
source extension, rules, 4-6
special characters, examples, 4-15
standard output, 4-18
starting, 4-17, 4-21
syntax, 4-17 - 4-18
tab character, 4-13
target extension, 4-12
target file creation, 4-17
target files, 4-3, 4-9
UNIX options not supported, 4-19
UNIX similarities and differences, 4-19
volume, 4-9
make built-in rules
 examples, 4-7
 .asm to .o rule, 4-7
 .c to .o rule, 4-6 - 4-8
make options
 -DDEBUG option, 4-14
 -t option, 4-14
 b option, 4-19
 d option, 4-19
 e option, 4-19
 i option, 4-19
 k option, 4-19
 m option, 4-19
 n option, 4-19
 q option, 4-19
 r option, 4-19
 s option, 4-19
 t option, 4-19
make, macros, 4-9
 examples, 4-13

AFLAGS, 4-11, 4-14
CFLAGS, 4-11, 4-14
command line, 4-12
makefile
 examples, 4-4 - 4-5, 4-20
 -f option, 4-3
 command line, 4-10
 definition, 4-3
 dependency, 4-10
 dependency entries, 4-3 - 4-5, 4-18
 executed commands, 4-3
 libc directory, 4-21
 misc directory, 4-23
 operating system commands, 4-3
 syntax, 4-15
 sys directory, 4-22
marking and returning, Z editor, 5-19, 5-50
matching patterns, Z editor, 5-14
mkarcv, make, 4-20
modifying text, Z editor, 5-21 - 5-22
movement, Z editor
 moving text, 5-23 - 5-25
 within C programs, 5-18
 within text, 5-8
 word movement, 5-17 - 5-18
multiple options, grep, 3-8

N

named buffers, Z editor
 advantage of, 5-24
 definition, 5-24
 moving text, 5-23
 yanking text, 5-24
notation conventions, 1-2
null character string, make, 4-11, 4-14

O

object files, make, 4-9
 removing, 4-21

operating system commands, make, 4-14
 makefile, 4-3
operator commands, Z editor, 5-52

P

paging commands, Z editor, 5-12
paragraph commands, Z editor, 5-51
path, make, 4-9
pattern searching, Z editor, 5-14 - 5-15
patterns, grep, 3-1
 special patterns, 3-4
 within a file, 3-2
 within multiple files, 3-2
positioning, Z editor
 line, 5-50
 within files, 5-50
prerequisite files, make, 4-3 - 4-4
Professional Level, 1-1
put commands, Z editor, 5-52

Q

QuikFix, z editor, 5-1
 -e option, 5-6 - 5-7
 -o option, 5-7
 :cc command, 5-36, 5-53
 :cf command, 5-36, 5-53
 :cn command, 5-36, 5-53
 :cp command, 5-36, 5-53
 :cq, 5-39
 :cq command, 5-36, 5-53
 :wq command, 5-39

R

range of lines, diff, 2-2
reading files command, Z editor, 5-41
redo commands, Z editor, 5-53
regular expression, Z editor, 5-14
 definition, 5-14
 special characters, 5-14

special characters list, 5-15
replace commands, Z editor, 5-51
replacing lines, diff, 2-2
requirements, Z editor, 5-2
return code, make, 4-15
rule-processing capability, make, 4-2
rules, make, 4-2, 4-12
 built-in, 4-6, 4-11, 4-14
 definition, 4-12
 makefile, 4-6
 overriding, 4-8
 sequence of commands, 4-6
 source extension, 4-6
 tab character, 4-13

S

screen, Z editor, 5-11
 adjusting, 5-49
 display, 5-3
 scrolling, 5-7
shift operators, Z editor, 5-26
shifting text, Z editor, 5-26
simple string matching, grep, 3-7
source extension, make, 4-12
standard input, grep, 3-3
standard output, make, 4-18
starting, make, 4-21
starting, Z editor, 5-38
status information line, Z editor, 5-3
stopping, Z editor, 5-39
string search commands, Z editor, 5-8, 5-12 - 5-14
substitute command, Z editor
 examples, 5-35
 options, 5-34
 syntax, 5-34

T

tab character, make, 4-13
tags, Z editor, 5-44

target extension, make, 4-12
 rules, 4-6
target file, Z editor, 5-25
target files, make, 4-3, 4-9
 creation, 4-17
technical support, 1-3
text, Z editor
 line length, 5-11
 rearranging, 5-24

U

undo command, Z editor, 5-26, 5-53
UNIX, 5-47
 diff, 2-4
 Ex-editor, 5-32
 grep, 3-7
 make, 4-1, 4-19
 Vi editor, 5-1
unnamed buffer, Z editor, 5-23
 deleting text, 5-23
 moving text, 5-23
unprintable characters, Z editor, 5-11

V

Vi editor, 5-47
volume, make, 4-9

W

wildcard characters, grep, 3-2 - 3-3
word movement commands, Z editor, 5-18, 5-51
wrap scan option, Z editor, 5-13
writing files commands, Z editor, 5-40 - 5-41

Y

yank operator command, Z editor, 5-23, 5-52

Z

Z editor

See also Z editor commands

.bak files, 5-6

accessing files, 5-39, 5-45

adjusting the screen, 5-20, 5-49 - 5-50

appending numbers to commands, 5-9

autoindent, 5-5, 5-28, 5-36

character positioning, 5-51

character strings, 5-14

colon commands, 5-13, 5-53

command mode, 5-6

components, 5-3

control keys, 5-12

corrections during insert, 5-52

creating new program, 5-3

ctags, 5-3, 5-45

cursor motion commands, 5-9, 5-18

cursor positioning, 5-7

definition, 1-2, 5-1

deleting lines, 5-10, 5-22

deleting text, 5-10, 5-21 - 5-22

differences between Z and Vi, 5-47 - 5-48

displaying unprintable characters, 5-11

duplicating text blocks, 5-23 - 5-24

echo commands, 5-12

editing another file, 5-41 - 5-43

editing existing files, 5-5

escape key, 5-5, 5-12

Ex-like commands, 5-32 - 5-35

exiting, 5-5

exiting insert mode, 5-5

file lists, 5-43

filenames, 5-40

indirect macro definition, 5-31

insert and replace, 5-51 - 5-52

insert command list, 5-26

insert commands, 5-10, 5-26 - 5-27

insert mode, 5-4, 5-27

insert mode, exiting, 5-12

line movement, 5-15 - 5-17
line positioning, 5-50
macro buffer, 5-28
macro wrap options, 5-32, 5-36
macros, 5-2, 5-28 - 5-32, 5-53
main text buffer, 5-23 - 5-24
making changes, 5-20
marking and returning, 5-19, 5-50
matching patterns, 5-14 - 5-15
miscellaneous operations, 5-52
modifying text, 5-21 - 5-22
movement within C programs, 5-18
moving text between files, 5-24 - 5-25
moving text blocks, 5-23
moving within text, 5-8
named buffers, 5-23 - 5-25
operators, 5-52
option file, 5-36 - 5-37
options, 5-45 - 5-46, 5-49
paging, 5-12
positioning within files, 5-50
QuikFix facility, 5-1
reading files, 5-41
rearranging text, 5-24
ree�行 macros, 5-31
regular expressions, 5-14
requirements, 5-2
screen display, 5-3 - 5-4
scrolling, 5-7 - 5-8
shift operators, 5-26
shifting text, 5-26
starting, 5-6, 5-37 - 5-38, 5-49
status information line, 5-3
stopping, 5-6, 5-39
string search, 5-8, 5-12 - 5-13, 5-30
substitute command, 5-34
tags, 5-44
text lines longer than screen, 5-11
undo and redo, 5-53
undo command, 5-26

UNIX vi similarities and differences, 5-47 - 5-48
unnamed buffer, 5-23 - 5-24
unprintable characters, 5-11
word movement, 5-17 - 5-18
words and paragraphs, 5-51
wrap scan option, 5-13
writing files, 5-40 - 5-41
yank and put, 5-52
yank operator, 5-23 - 5-24
yanking text, 5-24

Z editor command options

See also Z editor commands
ai=1/0, 5-49
ak, 5-49
bk=1/0, 5-49
cs, 5-46
eb=1/0, 5-49
ma=0/1, 5-49
sm=1/0, 5-49
ts=val, 5-49
wm, 5-32
wm=1/0, 5-49
ws=1/0, 5-49

Z editor commands

See also Z editor, special character commands

:se, 5-13
a/A, 5-10, 5-26 - 5-27, 5-51
b/B, 5-18, 5-30, 5-51
backspace, 5-9, 5-27
c/C, 5-21 - 5-22, 5-26, 5-52
carriage return, 5-9, 5-50
cc, 5-22
Control L, 5-20
Control-B, 5-12
Control-D, 5-7, 5-27
Control-F, 5-12
Control-H, 5-16
Control-J, 5-16
Control-K, 5-16
Control-L, 5-16

Control-U, 5-7
Control-V, 5-27
Control-X, 5-27
d/D, 5-21 - 5-22, 5-52
dd, 5-10, 5-20
dw, 5-22
 examples, deleting, 5-22
e/E, 5-18, 5-51
f/F, 5-17
fx/Fx, 5-51
g/G, 5-8, 5-50
H, 5-50
h/^H, 5-16, 5-51
i/I, 5-7, 5-26 - 5-27, 5-51
J, 5-52
j/^J, 5-16, 5-21, 5-50
k/^K, 5-16, 5-51
L, 5-50
l/^L, 5-16, 5-51
LF, 5-50
M, 5-50
mx, 5-19, 5-50
n/N, 5-9, 5-13, 5-50
o/O, 5-10, 5-17, 5-26, 5-52
p/P, 5-23, 5-25, 5-52
 examples, put, 5-25
R, 5-21, 5-52
rx, 5-21, 5-52
s/S, 5-21, 5-27, 5-52
space, 5-9, 5-51
t/T, 5-17
tabs, 5-49
tx/Tx, 5-51
u/U, 5-26, 5-53
v, 5-31, 5-53
w/W, 5-17 - 5-18, 5-30, 5-51
x/X, 5-7, 5-10, 5-20 - 5-21, 5-52
y, 5-23 - 5-24, 5-52
 examples, yank operator, 5-24
z/Z, 5-20, 5-49 - 5-50

ZZ, 5-5 - 5-6, 5-39
Z editor memory-resident buffer, 5-7
Z editor, command line options
 -e option, 5-6 - 5-7
 -o option, 5-7
Z editor, QuikFix facility
 See QuikFix, Z editor
Z editor, special character commands
 See also Z editor commands
 #, 5-40
 \$, 5-15, 5-17
 %, 5-18, 5-40
 *, 5-15
 /, 5-8
 <, 5-15
 <<, 5-52
 >, 5-15
 >>, 5-52
 ?, 5-13
 @, 5-11, 5-31
 [[, 5-18
 [], 5-14
 [^str], 5-15
 [str], 5-15
 [x-y], 5-15
 \, 5-40
]], 5-18
 ^, 5-15, 5-17
 {}, 5-19
 |, 5-17
 ~, 5-11
 comma, 5-17
 double quote, 5-19, 5-25
 period, 5-15
 semicolon, 5-17
 single quote, 5-19

ZOPT environment variable, Z editor, 5-36

Aztec C Source Level Debugger for the Amiga

**Version 5.0
October 1989**

Copyright 1988,1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

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

This software package and document are copyrighted ©1988,1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

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, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh 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

October 1989	Fifth Edition
December 1988	Fourth Edition
October 1988	Third Edition
July 1988	Second Edition
May 1988	First Edition

Table of Contents

Chapter 1 - Overview

Introduction	1 - 1
REQUIREMENTS	1 - 2
ABOUT THIS MANUAL	1 - 2
NOTATION CONVENTIONS	1 - 2
Technical Support	1 - 3

Chapter 2 - Getting Started

Introduction	2 - 1
Read the README File	2 - 1
Installation	2 - 2
Using the sdb Demonstration Programs	2 - 3
File Preparation	2 - 3
Startup	2 - 4
STARTING SDB	2 - 4
Where To Go From Here	2 - 6

Chapter 3 - Tutorial

Introduction	3 - 1
DISPLAYING SECTIONS OF THE SOURCE FILE	3 - 2
RUNNING THE PROGRAM	3 - 3
DISPLAYING THE TRACE OF CALLS	3 - 5
DISPLAYING VALUES AND COMPUTING EXPRESSIONS	3 - 6
WALKING UP AND DOWN THE FRAMES	3 - 7

DISPLAYING ASSEMBLY	3 - 7
---------------------------	-------

Chapter 4 - Commands

Overview	4 - 1
CATEGORIES	4 - 2
Basic	4 - 2
NAMES	4 - 3
Code and Data Symbols	4 - 3
Operator Usage of Names	4 - 3
LOADING PROGRAMS AND SYMBOLS	4 - 4
DEBUGGING LIBRARIES OR DEVICE DRIVERS	4 - 4
BREAKPOINTS	4 - 5
EXPRESSION BREAKPOINTS	4 - 6
TRACE MODE	4 - 6
BACKTRACING	4 - 7
MACROS	4 - 7
DISPLAYING SOURCE FILES	4 - 7
ABOUT WINDOWS IN SDB	4 - 8
Source and Data Displays	4 - 8
Command Display	4 - 9
Command Line Editing	4 - 9
Command Line History	4 - 10
Color Control	4 - 11
OTHER FEATURES	4 - 11
Use of Control Characters in a Command File	4 - 12
TERM DEFINITIONS	4 - 13
The Definition of expr	4 - 13
The Definition of addr	4 - 13
The Definition of RANGE	4 - 14

The Definition of CMDLIST	4 - 14
Detailed Description of Commands	4 - 14
AMIGA SPECIFIC COMMANDS	4 - 15
acc Show/set command window colors	4 - 15
acd Show/set data window colors	4 - 15
acs Show/set source windowcolors	4 - 15
LIST DISPLAY COMMANDS	4 - 16
add Display device list	4 - 16
adi Display interrupt list	4 - 16
adl Display library list	4 - 16
adp Display port list	4 - 16
adr Display resource list	4 - 16
am Display memory usage	4 - 17
ax Switch between main and alternate .dbg file	4 - 17
BREAKPOINT COMMANDS	4 - 17
bc Clear a single breakpoint	4 - 17
bC Clear all breakpoints	4 - 17
bd Display breakpoints	4 - 18
be Sets Expression Change Breakpoint	4 - 18
bm Sets Memory Change Breakpoint	4 - 19
br Resets breakpoint counters.	4 - 19
bs Set or modify a breakpoint	4 - 20
bt Toggle the call trace mode flag	4 - 20
bT Toggle the source line trace mode flag	4 - 20
DISPLAY COMMANDS	4 - 21
c Display Source Context	4 - 21
da Display local addresses	4 - 21
db Display memory in bytes	4 - 21
dw Display memory in words	4 - 21
dl Display memory in long words	4 - 21
d Display memory in last format	4 - 21
dc Display all code symbols	4 - 22

dd	Display all data symbols	4 - 22
df	Display source file lines	4 - 22
dg	Display global values	4 - 23
ds	Displays stack backtrace	4 - 24
dS	Displays stack, function arguments, and auto backtrace .	4 - 24
"FIND SOURCE STRING" COMMAND		4 - 24
/	Finds string in source file	4 - 24
FRAME COMMANDS		4 - 25
fu	Frame up command	4 - 25
fd	Frame down command	4 - 25
GO COMMANDS		4 - 26
g	Execute the program	4 - 26
G	Execute the program, without setting table breakpoints .	4 - 26
LOAD COMMANDS		4 - 27
ld	load device symbols	4 - 27
ll	load library symbols	4 - 27
lp	load program	4 - 27
Loading the Program		4 - 28
Loading the Symbol Table		4 - 28
MEMORY MODIFICATION COMMANDS		4 - 28
mb	Modify bytes of memory	4 - 28
mw	Modify words of memory	4 - 28
ml	Modify long words of memory	4 - 28
mc	Compare memory	4 - 29
mf	Fill memory with value	4 - 29
mm	Move memory	4 - 29
ms	Search memory	4 - 30
OUTPUT CONTROL COMMANDS		4 - 30
oe	Toggles command echo flag	4 - 30
op	Toggles pause flag	4 - 30
ow	Toggles switch to user screen	4 - 31
PRINT COMMAND		4 - 31

p	Generates formatted output	4 - 31
desc_code List	4 - 35	
Floating Point Number Print Codes	4 - 36	
Character Print Codes	4 - 37	
Special Purpose Codes	4 - 37	
P Command desc_codes For Setting Current Address	4 - 38	
QUIT COMMAND	4 - 39	
q	Quit the debugger	4 - 39
REGISTER COMMAND	4 - 40	
r	Register display	4 - 40
Description	4 - 40	
SINGLE STEP COMMANDS	4 - 40	
s	Single steps the program n times	4 - 40
S	Single steps into calls without display	4 - 40
t	Single steps n times stepping over calls	4 - 40
T	Single steps over calls, display last	4 - 40
UNASSEMBLE COMMANDS	4 - 41	
u	Unassembles memory, with symbols	4 - 41
U	Unassembles memory, without symbols	4 - 41
MACRO COMMANDS	4 - 42	
x	Create Macro Command	4 - 42
X	Display Macro Command	4 - 42
EXPRESSION COMMANDS	4 - 43	
=	Display the value of an expression in several formats ..	4 - 43
e	Evaluate an expression	4 - 43
REDIRECT COMMAND INPUT/OUTPUT	4 - 43	
<	Redirect command input	4 - 43
>	Log all I/O only	4 - 43
>>	Log commands only	4 - 43
HELP COMMAND	4 - 44	
?	List command	4 - 44

CHANGE MODE COMMAND	4 - 44
z Change mode command	4 - 44
DELAY COMMAND	4 - 45
& Delay sdb	4 - 45
Command Summary	4 - 46
AMIGA SPECIFIC COMMANDS	4 - 46
BREAKPOINT COMMANDS	4 - 46
DISPLAY COMMANDS	4 - 46
FIND SOURCE STRING COMMAND	4 - 47
CHANGE FRAME COMMANDS	4 - 47
GO COMMAND	4 - 47
LOAD COMMAND	4 - 47
MEMORY MODIFICATION COMMANDS	4 - 47
OUTPUT CONTROL COMMANDS	4 - 47
FORMATTED PRINT COMMAND	4 - 47
QUIT COMMAND	4 - 47
REGISTER COMMAND	4 - 48
SINGLE STEP COMMANDS	4 - 48
UNASSEMBLY COMMANDS	4 - 48
MACRO COMMANDS	4 - 48
EXPRESSION COMMANDS	4 - 48
REDIRECT COMMAND INPUT	4 - 48
HELP COMMAND	4 - 48
CHANGE MODE COMMAND	4 - 48
DELAY COMMAND	4 - 48

OVERVIEW

GETTING STARTED

TUTORIAL

COMMANDS

OVERVIEW

1

Chapter 1 - Overview

Introduction

Manx **sdb** is a fast, easy to use, interactive source level debugger. You will appreciate the simplicity of working with **sdb**. If you are an experienced programmer, you will recognize the time and effort you save debugging with **sdb**.

Use **sdb** to debug programs created with the Aztec C Compiler. The windows in **sdb** display C source and command output separately, with a third window for entering commands.

With **sdb**, you can...

- ...debug programs at the C source and assembly language levels*
- ...display all active function names*
- ...display values of passed parameters*
- ...examine variables from any active function*
- ...use function or line-by-line tracing*
- ...set breakpoints by lines, functions, or variables*
- ...see actual C source as it executes*
- ...customize the debugging environment with reusable command macros and procedures*
- ...and more!*

REQUIREMENTS

Manx recommends a minimum of 256K bytes of available RAM memory for use with **sdb**; the debugger uses about 180K. **sdb** runs under the CLI. **sdb** can be used on 68000, 68010, 68020, and 68030 systems.

ABOUT THIS MANUAL

This manual describes how to use the Source Level Debugger and is designed to be used primarily as a reference. Though this manual does include a brief tutorial, the best way to learn, in depth, how to use **sdb** is to work through the demonstration programs that have been included on your **sdb** disk. See the *readme* file or Chapter 2 of this manual for instructions on how to execute the demonstration programs.

Throughout this manual, we assume that you already have a basic understanding of how to use Aztec C. If you need additional technical information about using Aztec C, refer to your Aztec C documentation.

This manual is divided into the following sections:

Introduction	Describes the requirements and features of sdb .
Getting Started	Gives the procedures necessary to install sdb , start up sdb , and run the sdb demonstration program.
Tutorial	Gives easy-to-follow directions for using sdb .
Commands	Describes the features of sdb and how to use them. Defines some sdb terms and describes the sdb commands in detail. Includes a command summary for easy reference.

NOTATION CONVENTIONS

Throughout this manual, we use the following conventions:

input	to indicate data entered by the user (e.g., commands, options, and functions)
--------------	---

output	to show text that is generated by the computer
DEFINITION	small uppercase bold is used on terms that may be new to the user; most likely will include explanation or definition of term
{choice1 choice2}	braces and a vertical bar mean that you have a choice between two or more items
<i>placeholders</i>	information that must be supplied by the user, for example, <i>filename</i> , <i>range</i> , <i>identifier</i> , etc.
[<i>optional</i>]	to show optional information

Technical Support

At Manx, we build dependability into our products. However, if you should find that you need help, rest assured that we provide it.

Refer to your *Aztec C User Guide* for information on the Technical Support that Manx provides. If you have any problems with **sdb**, follow the procedures outlined in **Technical Information** chapter so that we may best be able to assist you.

GETTING STARTED

2

GETTING STARTED

Chapter 2 - Getting Started

Introduction

This chapter takes you through the steps necessary for getting started with **sdb**. Before you begin using **sdb**, you should:

- Read the **readme** file.
- Install **sdb**, as described in the **Installation** section of this chapter, on your hard drive OR create an **sdb** "work diskette"
- Run the **sdb** demonstration programs that have been included on your disk.
- Prepare your files for debugging.
- Startup **sdb**.

This chapter describes each of these procedures in detail.

Read the README File

If there is a **readme** file on your **sdb** disk, you can access it by entering "type **readme**" from the CLI. The **readme** documents provide information that

may not be in the manual or may be more recent than information in the manual.

If you want to send the **readme** document to the printer, use the command:

```
type > PRT: readme
```

Installation

Once you have verified that your files are complete, you should load **sdb** onto your hard disk or create a working floppy disk.

To load **sdb** onto your hard disk:

1. Turn on your machine and enter the CLI.
2. Insert your **sdb** disk into your floppy drive. Change directory to your hard disk's **bin** directory. Copy the file **sdb** into your **bin** directory using the command:

```
copy df0:sdb sdb
```

3. Change directory back to the root directory of your hard disk. Create a scratch directory and load the **sdb** examples into the scratch directory.

To create an **sdb** "work disk":

1. Turn on your machine and enter the CLI. Do not boot off of your Aztec C disk, as it does not give you access to CLI.
2. With the **sdb** master in drive zero, insert your work disk into drive one.
3. Copy the contents of your SDB disk to the work disk using the command

```
diskcopy df0: df1:
```

Once you have run through the tutorial and the examples, you may delete all but the **sdb** file from your work disk. You do not need any other files from the disk in order to use **sdb**.

Using the sdb Demonstration Programs

As explained in Chapter 1, the **Overview**, the best way to learn how to use **sdb** is to work through the demonstration programs that have been included on your **sdb** disk. After you have worked through the demonstration programs, you should also read the written tutorial, Chapter 3, which is included in this manual.

To start the demonstration, boot your machine into CLI. Do not boot from the SDB disk, as it does not make CLI available to you. As explained above, under "Read the **readme** File", it is a good idea to print out the **readme** file before beginning work. To begin the demonstration, change directory to the **demo** directory by typing:

```
cd demo
```

Then, simply type **demo**. After making sure that you want to proceed, the demonstration will "walk you through" the basics of using **sdb**. For a complete understanding of the power and flexibility available with **sdb**, you should work through each of the demonstrations included.

File Preparation

sdb works on executable files. To use **sdb**, however, you must create a debug file as well. The following steps will show you how this is done:

1. Create a C source file.
2. Compile using the **-bs** option, as shown:

```
cc -bs filename.c
```

3. Link using the **-g** option, as shown:

```
ln -g filename.o -lc
```

You will now have two files: an executable file and a debug file. The debug file will have the extension **.dbg**.

Note: To use **sdb**, you must have version 3.6 or higher of Aztec C.

Startup

You can start **sdb** in one of two ways:

1. From the CLI, by typing

sdb filename

2. From the CLI by invoking **sdb** and then using the **lp** command to load the program. (See the section on the **lp** command in the **Commands** chapter.)

The following sections discuss these startup methods.

STARTING SDB

When you start **sdb** from the CLI, as shown in step 1 above, you may also specify certain options and parameters, in the form of:

sdb [options] filename [parameters]

where

[*options*] are any one of the following:

-a starts **sdb** in assembly mode (default is C source mode; see **z** command in the **Commands** chapter.)

-cc_{o,t,b,p} set command area colors(*o*=outline, *t*= text, *b*=background, *p*=prompt). For each letter in the -cc, -cd, and -cs options, enter one of the numbers 0 through 3 to select one of the four workbench colors.

-cd_{t,b} set the data area display (*t*=text and *b*= background)

-cs_{t,b,ht,bt} set the source display colors (*t*=text, *b*=background, *ht*= highlighted text, and *bt*=highlighted background)

-cwl,t,w,h	controls the positioning and size of the sdb window. All four numbers must be present; they determine (in order) left edge, top edge, width, and height.
-d#	specifies data window buffer size with valid range of $512 < \# < 32000$ (default is 4096 bytes).
-e	toggles echoing of commands to the data window (see the oe command in the Commands chapter.)
-h#	specifies history window buffer size with valid range of $128 < \# < 10240$ (default is 512 bytes).
-mfile	forces redirection to an input <i>file</i>
-n	does not look for <i>file.mac</i>
-p	sets prompt for -w mode
-r	use this option if the program being debugged has been made resident using the REZ utility.
-spath	path is a string to be prefixed to source files upon opening
-w	disables the windowing part of the debugger display

and [*parameters*], like *arg1*, *arg2*..., are character strings to be passed to the program. When invoked, **sdb** loads the program to be debugged, starts it, and then stops at the entry point to the main function displaying that line.

The startup options can be changed and made somewhat permanent by setting the environment variable **SDBOPT** to the appropriate values. For example, to preset the colors for the three windows, enter the following:

```
set "SDBOPT=-cs1,3,2,0 -cd3,2 -cc0,0,1,2"
```

If the program *filename* specifies a drive or directory, **sdb** searches for the program file in only that particular area. Otherwise, it searches the current directory on the default drive.

Arguments are passed to the program using the **argv** parameters of the program's main function: **arg1** is pointed at by **argv[1]**, **arg2 argv[2]**, and so on. **argv[0]** contains the name of the program.

-spath creates the fpath macro, which tells **sdb** to look for source files in a sequence of directories. By default, **sdb** looks in the current directory. If you do not specify another directory or subdirectory, **sdb** assumes the current directory should be searched. However, you may prefer to separate your executable files from your source files. Therefore, **sdb** searches multiple directories as long as you specify the directories or subdirectories on the command line, separated by a ";" or "!"'. For example,

```
-sdir1;vol:dir2;...
```

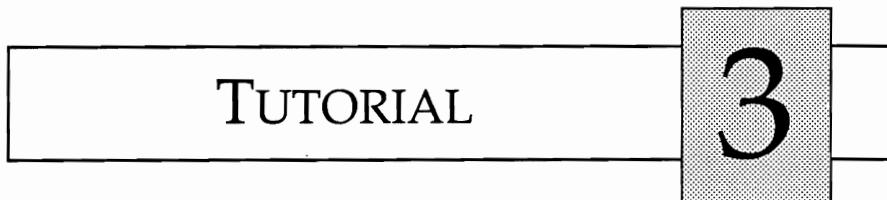
or

```
-sdir1!vol:dir2!...
```

You may also invoke **sdb** by typing **sdb** with no arguments. You must use the **lp** command to load the program you want to debug.

Where To Go From Here

Now that you have worked through the steps necessary for getting started with **sdb**, you should go on to the **Tutorial**, Chapter 3.



Chapter 3 - Tutorial

Introduction

This tutorial introduces you to the Source Level Debugger. As explained in the previous chapter, **sdb** works on executable files. To use **sdb**, however, you must create a debug file as well. Refer to the **File Preparation** section of Chapter 2, **Getting Started**, for instructions on how this is done.

With this tutorial, we will assume that you already have a basic understanding of the Aztec C software development system and how it is used. If you have any questions about using Aztec C, refer to your Aztec C documentation.

When using the debugger, you should make use of the help screens, which can be obtained by entering

?

Further detail is provided for many of the commands by entering the first letter of the command followed by a question mark. For example, typing

f?

will display information on the commands **fu** and **fd**.

To begin this session, you must invoke **sdb**. Refer to the **Startup** section of Chapter 2, **Getting Started**, for instructions on how this is done. As explained in that section, you may enter **sdb** from the CLI.

When invoked, **sdb** loads the program to be debugged, starts it, and then stops at the entry point to the main function, displaying that line.

DISPLAYING SECTIONS OF THE SOURCE FILE

Two commands, **c** and **df**, are provided for displaying your source file.

c is used to display the current source line. It takes no arguments.

df can be used to display lines other than the current source line. The format of the **df** command is:

df [filename] range

where *filename* is the name of the file to be displayed and *range* is the line number that will be displayed at the beginning of the source window. Note that *filename* is optional and defaults to the current file if none is specified.

range is one of the following:

line
line ... line
line, count

line and *count* are expressions yielding an integer result. If *line* is specified alone, the line indicated is displayed at the beginning of the source window.

The **c** command is very useful if you have used **df** to display another file or have scrolled out of the source window. If this is the case, then typing

c

will bring you back to your current file at the current line.

As with all display commands, pressing <CR> after issuing the command causes the next set of lines of source to be displayed.

RUNNING THE PROGRAM

Once the program is loaded, debugging consists of setting breakpoints and running the program. Breakpoints can be one-time breakpoints or can be set as permanent breakpoints.

If you do not wish to set permanent breakpoints, you can control execution of the program using the single step and `go` commands.

The format of a single step command is:

`[count]s`

`[count]S`

`[count]t`

`[count]T`

where *count* is a positive integer that causes the debugger to single step *count* times, printing information for each breakpoint before stopping.

If *count* is zero, single stepping is performed until a breakpoint is hit, an error occurs, or the program finishes.

The difference between `s` and `t` is that `s` single steps *into* calls while `t` single steps *across* calls. In effect, `t` treats a call as a single line.

`S` and `T` step *count* times, displaying information only for the last breakpoint taken. `S` steps into calls just like `s` while `T` treats calls as a single line.

In source level mode (the default), single step is by source line. In assembly mode, (set by typing `z<CR>`) single step is by machine code instruction.

If single stepping encounters a function for which no source line information is available (a library function for example), single stepping into the function causes the debugger to step over the call just as it would with `t`.

To set a permanent breakpoint, type:

[count]bs addr[;command]

where *count* is an integer expression, *addr* is:

[filename].line

function[.line]

addressexpression

and *command* is a set of debugger commands separated by semicolons.

For example, you can enter:

bs linkmain.c.39

This sets a breakpoint on line 39 of **linkmain.c**. Alternatively,

bs getfld.10

sets a breakpoint at line 10 in the function **getfld**.

bs getfld+10

sets a breakpoint at 10 bytes beyond the start of **getfld**. Entering

bs .10

will set the breakpoint at line 10 in the file currently being displayed.

Finally, to make the program go, you simply type **g<CR>**. The program executes until it encounters a breakpoint or terminates.

The format of the go command is:

g [@] [addr]

where @ means "go until a return after *addr* is found" and *addr* is as described above. *addr* is set as a temporary breakpoint and will not be remembered after execution of the command.

For example:

g @

means go until the current function returns. And

```
g linkmain.c.39
```

means go to line 39 in `linkmain.c`

DISPLAYING THE TRACE OF CALLS

It is often useful to know where the program is in a set of nested calls and what the arguments and local variables are to each of the calls. To display the nested stack of calls in `sdb`, simply type

```
ds
```

Each of the calls currently active is displayed, together with its arguments; each argument is displayed in a form appropriate to its type.

Type

```
ds
```

and the functions are displayed with the types, names, and values of each function argument, and auto variables.

For example:

```
ds
```

might cause the following display

```
_main(int argc=1, char ** argv = 0x00C051B8)
__main (0x0000, 0x0002, 0x00C0, 0x5040
.begin ()
while
```

```
ds
```

could cause:

```
_main(int argc = 1, char **argv = 0x00C051B8)
    int i = 0
    long j = -1
    char name[8] = "hello."
__main()
.begin
```

DISPLAYING VALUES AND COMPUTING EXPRESSIONS

sdb provides some simple but powerful facilities for displaying variables, arrays, and structures. These are the **p** (print) and the **e** (evaluate) commands. For example, to print a structure named **symbol1**, you simply enter:

```
p symbol1
```

which might result in:

```
struct symbol1b symbol = {  
    int s_flag = 10  
    char *s_name = 0xFF42  
    int s_value[2] = {  
        10,5  
    }  
}
```

Suppose then you want to display the string pointed to by **symbol1.s_name**. You simply type:

```
ps *symbol1.s_name
```

The **s** following the **p** command indicates that you are overriding the default format for pointer to **char** and the **s** indicates that the new format is string.

So the result might be:

```
pointer
```

In addition to the print command, the debugger has an evaluate command so that you can perform general C expression evaluation, including calls to C functions, assignment, pre- and post-increment and decrement, casts, and conditionals.

```
e c = getchar()
```

might result in:

=10

WALKING UP AND DOWN THE FRAMES

In using the expressions shown above, you are generally limited to referring to variables that are visible by C rules at the point where execution stopped. Therefore, you cannot refer to local variables or functions that are not active or are not the "current" function. (Statics, however, may be referred to by qualifying them with the name of the file or function they were declared in, e.g. `linkmain.c.name` or `main.name`.)

In order to refer to names in other active functions, you can change `sdb`'s notion of what the current function is by walking up and down the call frames, using the commands `fu` for frame up, and `fd` for frame down. These commands walk up the call frames, displaying the line from which the next frame down was called and making visible all of the current call's local variables.

DISPLAYING ASSEMBLY

Finally, you can display the assembly code at any address with the `unassemble` command. Its format is:

u *addr*

U *addr*

where *addr* is as described above.

u does a disassembly with symbols substituted where possible for global and local variables. **U** disassembles without symbol substitution but with the hex for the code shown as well as the assembly.

COMMANDS

4

Chapter 4 - Commands

Overview

sdb commands consist of one, two, or three characters. The first identifies the command category and the second and third identify the specific operation to be performed. If the command name contains only one character, it is the only command in a category.

If you are using Motorola Fast Floating Point, use sdbf. sdbf is identical to sdb except for the support of Motorola fast float.

This chapter contains the following:

- | | |
|-----------------------------|--|
| Categories | describes the general command categories |
| Detailed Description | describes how to use sdb and sdbf commands in detail |
| Command Summary | lists the commands with a brief description. |

CATEGORIES

Basic

sdb has two types of commands for examining memory: display (d) and print (p). The display commands db, dw, and dl simply display hexadecimal bytes, words, and long words, respectively.

The print command, **p**, is more powerful, allowing you to print variables, arrays, and structures by name without having to worry about data types. For example, you can tell **sdb** to print a structure whose name is **symbol** by typing

```
p symbol
```

sdb figures out the data types and prints the structure and its members in the proper format. You can also display strings that are pointed to by a structure member using the **p** command.

The evaluate command, **e**, allows you to perform general C expression evaluation, including calls to C functions, assignment, pre- and post-increment and decrement, casts, and conditionals.

The register command, **r**, displays and modifies the 68000 registers.

The frame commands, **f**, allow you to walk up and down the call frame.

The memory modify commands, **m**, allow you to modify memory.

The unassemble commands, **u**, display code symbolically, in a form similar to its appearance in an assembly language source file.

The **s**, **t**, and **g** commands cause your program to be executed. **s** and **t** commands "single step" through your program, e.g., they execute a specified number of instructions in your program and then return control to **sdb**.

The **g** commands transfer control of the processor unconditionally to your program. In this case, **sdb** regains control when your program terminates, when an error occurs (such as division by zero), or when a "breakpoint" is taken. (Breakpoints are described in more detail later in this chapter.)

The help command, **?**, displays a summary of all **sdb** commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category followed by a **?**. For example, type

```
m?
```

to get information about the memory modification commands (all of which contain a first letter **m**).

NAMES

sdb allows memory locations to be referenced by name as well as by location. It learns a program's variable names by reading the file containing the program symbol table.

The Linker generates a symbol table file for a program in response to option **-g**. There are various ways for **sdb** to read the symbol table and these are described below.

sdb allows global symbols, automatic variables, static variables, arrays, and structures to be accessed by name.

Code and Data Symbols

sdb classifies symbols as being either code or data symbols, e.g., they refer to a location in a physical code segment or a physical data segment.

There are two commands for viewing the symbols that are known to **sdb**: **dc** and **dd**, which display code and data symbols, respectively.

Operator Usage of Names

When a C source program is compiled, all global symbols are truncated to a maximum of 31 characters and are then prepended with an underscore character.

To refer to symbols which, in a C source file, contain less than 31 characters, the inclusion of the prepended underscore character is optional. If it is specified, **sdb** searches for that symbol in its symbol table. Otherwise, it first searches for the specified symbol. If the search fails, it prepends an underscore to the name and searches again. To refer to symbols that contain 32 or more characters, only the first 31 are significant. **sdb** ignores all other characters in the name.

LOADING PROGRAMS AND SYMBOLS

A program and its symbols can be loaded into memory when **sdb** is started. In this case, the command line defines the program to be loaded.

The **sdb load** program command , **lp**, can also be used. If **sdb** is started with a program name specified on the command line, the **lp** command can

LOADING PROGRAMS AND SYMBOLS

A program and its symbols can be loaded into memory when **sdb** is started. In this case, the command line defines the program to be loaded.

The **sdb load** program command , **lp**, can also be used. If **sdb** is started with a program name specified on the command line, the **lp** command can be used to reload the same program for another debugging session. If **sdb** is started without specifying a program name, the **lp** command must be issued with the desired program name. Only one user program can be in memory at once. When told to load a program, **sdb** automatically tries to load the program symbol table too. It assumes the symbol table file has the same name as the program file, with the extension changed to **.dbg**.

When a program exits, it must be reloaded with the **lp** command before execution can begin again.

DEBUGGING LIBRARIES OR DEVICE DRIVERS

Because a library or device driver is not “run” in the same way as a program, **sdb** contains special support to enable these to be debugged at the source level. First, they must be compiled with option **-bs** and linked with option **-g**.

Then the binary generated by the linker should be placed in the **libs**: directory while the **.dbg** file should be in the source directory. Next the program that will make calls to the library or device must also be compiled and linked to generate source level information.

Now, **sdb** should be invoked on the test program. When the program is loaded, it should be run far enough so that the appropriate library or device is in memory. Use the **adl** or **add** command to verify this. Then the **ll** or **ld** command should be used with the appropriate library or device name as in:

```
ll mylib
```

or

```
ld mydev
```

BREAKPOINTS

Before transferring control of the processor to your program in response to a **g** command, **sdb** can set **BREAKPOINTS** at specified locations in the code. When your program reaches a breakpoint, **sdb** regains control.

A breakpoint has a skip count associated with it that allows it to be passed several times before actually taking the breakpoint and returning control to **sdb** and the user. When a breakpoint is reached, **sdb** is always activated. It increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken. For example, **sdb** retains control of the processor. Otherwise, **sdb** returns control of the processor to your program after the breakpoint. By default, a breakpoint's skip count is 0. Thus, each time the breakpoint is reached, it is taken.

A breakpoint can also have a sequence of **sdb** commands associated with it. When a breakpoint is taken, these commands are executed before **sdb** allows you to enter commands. For example, if you want only to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location and specify a list of commands to do just that. The first command in the sequence would be a **d** command to display memory, and the second would be a **g** command to continue execution of the program.

There are two ways to define breakpoints: with the **g** command and with special breakpoint commands, whose first letter is **b**.

The breakpoint commands manipulate a table of breakpoints: There are commands for entering breakpoints into the table, displaying the entries, resetting their counters, and removing them from the table.

There is a difference between a breakpoint defined in a **g** command and those in the breakpoint table: The **g** command breakpoint is temporary, while a breakpoint table is more permanent (it exists until removed from the table). Before transferring control to your program in response to a **g** command, **sdb** sets all breakpoints that are in the breakpoint table and that are specified in the **g** command itself. When a breakpoint is taken, **sdb** removes all breakpoints from the code and forgets all about the **g** command breakpoint. The breakpoint table breakpoints, however, are still in the table and will be set back in memory when control is again returned to your program.

sdb remembers the skip counter associated with a breakpoint in the breakpoint table. When it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (e.g., its value in the table). When a breakpoint is taken, the accumulated count for the breakpoints in memory is saved in the breakpoint table.

EXPRESSION BREAKPOINTS

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called an expression breakpoint, is taken when a specified expression evaluates as true.

With an expression breakpoint set, **sdb** detects the instruction that causes the specified expression to evaluate as true, assuming your program is being single stepped using an **s** command.

When an **s** command is used to single step a program and an expression breakpoint is set, **sdb** evaluates the specified expression after each instruction is executed and takes a breakpoint when appropriate.

The **be** command is used to set and remove expression breakpoints.

TRACE MODE

sdb supports two trace modes that display information whenever a function is entered or exited or when a source line is passed.

With the first mode enabled upon entry to, and exit from, a function, the function name, its arguments, and return values are displayed.

The commands **bt** and **bT** affect trace mode: **bt** enables and disables call trace mode, and **bT** enables and disables source line trace mode.

BACKTRACING

When **sdb** regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location: The **ds** and **dS** commands display information about the currently executing function, and the func-

tion that called it, and so on, back to the Manx function `_main()`, which called your function `main()`.

For each function, `ds` displays its name, arguments that were passed to it, and the address to which it will return. `dS` displays the function's automatic variables as well.

MACROS

`sdb` allows you to define and execute macros, e.g., a sequence of `sdb` commands.

Macros are written to a file that is saved so that macros can be reused in a subsequent session. The filename for the saved macros is derived by taking the program name and appending a `.mac`. The `sdb` command `x` is used both to define and execute a macro.

DISPLAYING SOURCE FILES

Since `sdb` is a source level debugger, it allows you to display source files, thus providing a convenient means to examine the source of a program being debugged.

Only a single source file can be examined at a time. The display source command, `df`, and the context command, `c`, can be used to display its lines.

The find string command, `/`, finds a character string in the source file.

ABOUT WINDOWS IN SDB

`sdb` provides a single independent window created in the Workbench screen. The title on the window is "Aztec SDB:", followed by the name of the source file being displayed and the current function name in which the debugger has stopped.

Within this window are three independent display areas: a source area, a command area, and a data area. The entire window can be resized using the sizing icon located in the lower right corner of the window. The minimum size guarantees two source lines, the command line, and three

data lines. Lines that are too long to fit within the source or data displays are truncated, which is indicated by a \$ character in the last column.

Source and Data Displays

Both the source and data areas operate similarly. Each can be considered to be a window containing text or data from a larger set of text or data. The position of this window on the larger set is controlled by an icon at the right side of the display. This gadget consists of a rectangle with a smaller square within it, bordered at the top and the bottom with two arrows pointing up and down, respectively. The arrows are used for fine adjustments of the display, while the squares are used for larger adjustments.

The outer rectangle of the gadget represents the total amount of text or data that can be displayed. The inner square, by its position, represents the position of the displayed text within the total. To reposition the window, move the mouse pointer over the scroll box and move the inner square up or down, while holding down the mouse button. As the square is moved, the display will be updated with the contents appropriate to the new position. When the desired position is reached, release the mouse button.

To move the display only one display's worth at a time, point the mouse either above or below the inner square but within the outer rectangle. Now, each mouse click will move the display up or down by the number of lines within the display.

To move the display one line at a time, point the mouse at one of the two arrows and click the mouse button. The display scrolls for each click of the mouse button. It scrolls continuously if you hold the mouse button down.

The displays may also be scrolled using the UP and DOWN cursor keys on the keyboard. To move the source display, hold down either SHIFT key while pressing the UP or DOWN cursor keys. The data display is similarly controlled by using the ALT key instead of the SHIFT key.

The source display is a window on the current text file. The data display is a circular buffer of 2000 characters. As lines are added to the end of the display, lines are removed from the beginning of the display. Because the buffer is character based, the display can show a few long lines, many short lines, or something in between.

Command Display

The command display is a single line display that consists of a **CMD?** prompt, an editing area, and a small UP/DOWN gadget to the extreme right.

The UP/DOWN gadget is used to position the command display within the **sdb** window. When the mouse is pointed at the gadget and the button is held down, an outline of the command area is displayed that follows the position of the mouse pointer. When the button is released, the command line is moved to the new position. This automatically causes the sizes of the source and data displays to be adjusted, and their contents and controls are redisplayed to reflect the new sizes.

The command display can also be moved by holding down the SHIFT and ALT keys simultaneously while pressing the UP and DOWN arrow keys.

Command Line Editing

When the command display is awaiting input, a block cursor is displayed within the editing area of the command display. At this point, characters may be typed into the display area using the keyboard. As each key is typed, it is inserted and the cursor moves to the right. When the BACKSPACE key is pressed, the cursor moves left one character and that character is deleted. A line can be up to 128 characters in length. If the line exceeds the width of the display, it will automatically scroll to the left as characters are added. When a line is finished, pressing the return key causes the line to be parsed and executed by **sdb**.

In addition to the BACKSPACE key, there are a number of editing features that are available within the command display. First, there are a number of ways to change the cursor position. Use the LEFT and RIGHT arrows to move the cursor one character position in the corresponding direction. Hold down either SHIFT key and press the LEFT or RIGHT arrow key to move the cursor to the beginning or end of the line. And point the mouse anywhere in the line and press the mouse button to move the cursor to that spot.

Once the cursor is positioned, characters can be deleted at the current position by using the DEL key. New characters can be inserted by simply typing them. Space is made for each character as it is typed.

If instead of inserting characters, the new characters are to replace the existing characters, press ESC, to switch from insert mode to replace mode. Press ESC again to switch back to insert mode.

Command Line History

When the RETURN key is pressed to signal the end of an input line, the line is added to the end of a 512-byte buffer. Longer lines mean fewer lines in the buffer, while shorter lines mean more lines in the buffer.

When a line is to be added to the buffer, it is first searched to see if the same exact line already exists in the buffer. If not, then the new line is simply added to the end and lines from the beginning are removed to make space. If it is found, the old line is simply moved to the end of the buffer after moving all the intervening lines. In this case, no lines are removed from the beginning. This is done so that simply typing s followed by RETURN a hundred times while stepping through a program will not flush all the old lines out of the buffer.

For each press of the UP key, the command line shows the previous line in the history buffer. When the desired line is found, it can be entered by pressing RETURN immediately or it can be edited as discussed in the previous section.

Use these additional character key sequences to move as follows:

F10 to swap screens between your program and **sdb**

ESC to toggle insert versus overstrike mode

CTRL-C to terminate output

CTRL-X to clear the command line

Color Control

Since the **sdb** window is created in the Workbench screen, **sdb** is limited to the four Workbench colors selected using the Preferences program. However, there is full control of the colors used within each display area. The Workbench colors are referred to by numbers 0 to 3. These correspond to the four colors displayed in order by the Preferences program.

The data display has the simplest choice of colors, namely the color used to display text and the color of the background. The default is 1,2 or text set to color 1 and background set to color 2.

The source display has four choices of colors. The first two correspond to the color and background of text similar to the data display. The second two correspond to the color and background of highlighted text. Highlighted text is used to show the current position of the program within the file. The default is 1,0,1,3 or text set to color 1, background set to color 0, highlighted text set to color 1 and highlighted background set to color 3. To display highlighted text as the inverse of regular text, simply reverse the text and background values as in 1,0,0,1.

Finally, the command display area has four choices. Just like the source and data displays, the text and background can be chosen. In addition, the text color of the CMD? prompt can be chosen as well as the color of the outline of the command area. The outline consists of a one pixel high line above and below the command area. The default is 1,1,0,1 or an outline in color 1, text in color 1 with a background of color 0, and the prompt set to color 1 as well.

The display colors can be examined and changed by using the ac commands in **sdb**. In addition, the colors can also be set using the -cs, -cc, and -cd options to **sdb**. To make the changes somewhat automatic, the environment variable SDBOPT is scanned for arguments in addition to those used to invoke **sdb**.

OTHER FEATURES

Some other features of **sdb** that have not yet been discussed are:

- redirect **stdin** command, <, causes **sdb** to read commands from a specified device or file and then continue reading commands from the console. Log commands, >, and >>, allow the logging of either all I/O or the commands only to a separate file.
- evaluate expression command, =, evaluates an expression.
- help command, ?, lists commands with a brief description of each.

Use of Control Characters in a Command File

Most of the keyboard commands have control character equivalents. These may be used directly from keyboards that support the use of control characters, but their primary use is for command files. This can be seen in the command file scripts used to run the sdb demos found on your disk. The following are the control characters and their keyboard equivalents.

Control Character	Keyboard Sequence	Action Performed
^K	up arrow	scroll back in history buffer
^D	down arrow	scroll forward in history buffer
^G	left arrow	move cursor left
^L	right arrow	move cursor right
^T	shift - up arrow	scroll C source down
^U	shift - down arrow	scroll C source up
^V	shift - left arrow	move cursor to far left
^W	shift - right arrow	move cursor to far right
^N	ALT - up arrow	scroll data down
^D	ALT - down arrow	scroll data up
^[F10	view application screen
^Y	ESC	toggle between Insert and Replace mode

TERM DEFINITIONS

This section defines some terms that are used in the command descriptions.

These terms are *expr*, *term*, *addr*, *range*, and *cmdlist*.

The Definition of *expr*

An *expr* is any valid C expression.

For example, an *expr* can consist of a single *term*, a series of *terms* separated by operators, the use of registers by their standard names, or 32-bit values representing memory locations.

Here is an example of *expr*:

```
si  
x + 2.0  
(i == j) ? 2 : 3  
0xC0B54  
sin(y)  
array[i]
```

The Definition of *addr*

An *addr* is the name of a C variable, an address constant, or a C expression that yields an address. Examples of an *addr* are as follows:

<code>linkmain.c.19</code>	address of the code corresponding to line 19 of <code>linkmain.c</code>
<code>.19</code>	same as above if current file is <code>linkmain.c</code>
<code>main</code>	address of <code>main</code>
<code>0x1532</code>	an address at <code>0x1532</code>
<code>token[j]</code>	address of <code>j</code> th element of array <code>token</code>

The Definition of RANGE

A *range* defines a block of memory. It has one of the following forms:

addr,cnt

addr to addr

(the word "to" must be included)

addr

,cnt

The form *addr,cnt* specifies the starting address, *addr*, and a number, *cnt*. *cnt* is interpreted differently by different commands. For example, the disas-

semble code command, **u**, displays *cnt* lines, while the display bytes command, **sdb**, displays *cnt* bytes.

The form *addr to addr* specifies the starting and ending addresses of the range. A full range need not be explicitly specified, because **sdb** remembers the last-used range and sets unspecified RANGE parameters from the remembered values:

- When a *range* is specified that consists of a single *addr*, the last used *cnt* is used.
- When a *range* is specified that consists of ,*cnt*, the next consecutive address is used, and the remembered count is changed to the new value.
- When nothing is specified as the *range*, the next consecutive address is used as the starting *addr*, and the *cnt* is set to the remembered value.

The Definition of CMDLIST

A *cmdlist* is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

command [*;command* ...]

If a macro is in a *cmdlist*, it must be the last command in the list.

Detailed Description of Commands

The commands are listed alphabetically within categories.

AMIGA SPECIFIC COMMANDS

- acc Show/set command window colors
- acd Show/set data window colors
- acs Show/set source windowcolors

Syntax

```
acc [out, text, back, prompt]  
acd [text, back]  
acs [text, back, high, back]
```

Description

The **acc**, **acd**, and **acs** commands control the color for the command data and source displays, respectively. See the **Overview** section of this manual for a detailed description of "windows" and "color codes." Because **sdb** is created in the Workbench screen, it is limited to the four Workbench colors selected using the Preferences program. Choose the four Workbench colors by referencing numbers 0 through 3.

acc: You have four color choices for the command display—text, background, prompt text, and command area outline. The default values are:

1, 1, 0, 1

where

Outline	Color 1
Text	Color 1
Background	Color 0
Prompt Text	Color 1

acd: You have two color choices for the data display: text and background. The default values are:

1, 2

where

Text	Color 1
Background	Color 2

acs: You have four color choices for the source display: the first two correspond to the color and background of text, the second two correspond to the color and background of highlighted text. (Highlighted text is used to show the current position of the program within the file.) The default values are:

1, 0, 1, 3

where

Text	Color 1
Background	Color 0
Highlighted Text	Color 1
Highlighted Background	Color 3

LIST DISPLAY COMMANDS

- add Display device list
- adi Display interrupt list
- adl Display library list
- adp Display port list
- adr Display resource list

Syntax

add
adi
adl
adp
adr

Description

These commands display the addresses and names of one of the various lists pointed to by ExecBase in the Amiga.

- **am** Display memory usage

Syntax

am

Description

This command displays the amount of free memory in the system.

- **ax** Switch between main and alternate .dbg file

Syntax

ax

Description

This command is used to switch back and forth between the two “personalities” of **sdb**—that which knows about the source and symbols of the test program discussed earlier in this **Command** section, and that which knows about the source and symbols of the library or device driver.

BREAKPOINT COMMANDS

- **bc** Clear a single breakpoint
- **bC** Clear all breakpoints

Syntax

bc *addr*
bC

Description

These commands delete breakpoints from the breakpoint table. **bc** deletes the single breakpoint specified by the address *addr* and **bC** deletes all breakpoints from the table.

► **bd** Display breakpoints

Syntax:

bd

Description

This command displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

- Its address, using a symbolic name, if possible
- The number of times it has been “hit” without a breakpoint being taken
- The skip count for it
- The command list for it, if any.

For example, a **bd** display might be:

# - Address	hits	skip	command
C1 - c62f9c _printf	1	2	
2 - c62b28 prog.c.19	0	0	p i

In this example, two breakpoints are in the table. The first is at the beginning of the function **_printf**; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the next time **_printf** is reached.

The second is on line **19** of the file **prog.c**; a breakpoint will be taken each time this line is about to be executed and will print the value of the variable “**i**”.

► **be** Sets Expression Change Breakpoint

Syntax

be [expr]

Description

This command is used to set and clear an expression breakpoint. This breakpoint is set by entering an arbitrary C expression. This expression is then evaluated each time a function is entered or exited. Whenever it is true, the breakpoint is taken, otherwise not. To cause the expression to be evaluated on each C source line, you must be running in single step or trace mode. This normally is not desirable since it slows down execution speed. To clear the breakpoint, use **be** with no arguments.

- **bm** Sets Memory Change Breakpoint

Syntax

bm [*range*]

Description

This command is used to set or clear memory change breakpoints. It does a checksum on the specified range on function entry and exit when the call trace mode is active (see **bt**), or on each instruction when the source line trace mode is active (see **bT**) or when running in single step or trace mode. The best way to locate a memory change is to first use the call trace mode to narrow it down between function calls. Then use the source line trace or single step mode to find the actual instruction.

When using single step mode in locating a memory change, enter the memory change breakpoint and then enter a large numeric prefix followed by an S or T. This will allow **sdb** to continue to check the location after each instruction and at the same time not require any input from you until the memory location has changed and the breakpoint activated or the specified number of instructions have been executed.

To clear a memory change breakpoint, enter **bm** with no range.

- **br** Resets breakpoint counters.

Syntax

br [*addr*]

Description

The **br** command resets the hit counter for the specified breakpoint which is at the address *addr*. If *addr* is not given, the hit counters for all breakpoints in the breakpoint table are reset.

- **bs** Set or modify a breakpoint

Syntax

[#] **bs** *addr* [*;command*]

Description

The **bs** command enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter **#** is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached, it will be taken.

The optional parameter *cmdlist* is a list of debugger commands to be executed when the breakpoint is taken.

- **bt** Toggle the call trace mode flag
- **bT** Toggle the source line trace mode flag

Syntax

bt
bT

Description

The **bt** and **bT** commands toggle the call trace mode and source line trace mode flags, respectively. The state of the trace mode flag determines whether trace mode is enabled or disabled.

In call trace mode, the debugger prints the names and arguments of each call within the program as it executes. On return, the value of the function's return is printed.

In source line trace mode, each statement of the program is displayed before it is executed.

DISPLAY COMMANDS

- **c** Display Source Context

Syntax

c

Description

This command centers the active source line in the source window.

- **da** Display local addresses

Syntax

da
dA

Description

The **da** command displays the automatic variables of the current function, while **dA** displays their addresses.

- **db** Display memory in bytes
- **dw** Display memory in words
- **dl** Display memory in long words
- **d** Display memory in last format

Syntax

db [range]
dw [range]
dl [range]

Description

The **db**, **dw**, and **dl** commands display successive bytes and words of memory, respectively.

d displays memory using the last format specified. For example, if **d** is entered, and **db** was the last display memory command, then **d** displays bytes, too.

The starting address of the *range* parameter is optional; if not specified, it defaults to the ending address of the last display's *range*, plus one. Each line of the display begins with the segment and address, followed by a hexadecimal display of 16 bytes or 8 words, followed by an ASCII display, by bytes, of the same data.

For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

- **dc** Display all code symbols

Syntax

dc

Description

The **dc** command lists all the code symbols in the memory-resident symbol table. For each symbol, its name and address are displayed.

- **dd** Display all data symbols

Syntax

dd

Description

The **dd** command lists all the data symbols in the memory-resident symbol table. For each symbol, its name and address are displayed.

- **df** Display source file lines

Syntax

df [filename] range

Description

The **df** command displays lines from the source file that was specified by the *filename* parameter.

The *range* parameter specifies the numbers of the lines to be displayed.

The starting line number is optional; if not specified, the display starts with the current line.

The current line in a source file is set by the source file commands **df** and **/**, as follows:

- When the file is first loaded with the **df** command, the first line in the file is the current line
- When the last source file command was **DISPLAY SOURCE**, **df**, the current line is the line following the last one displayed
- When the last source file command was **FIND STRING**, **/**, the current line is the line in which the string was found

The **df** command also sets the F-DOT for the source file to the number of the first line displayed. The F-dot is the line referred to when the starting line number of the range in a **df** command specifies a period (.). Also, source string searches begin at the line following the F-dot line.

Each displayed line is preceded with a line number in decimal, a colon, and the line itself.

► **dg** Display global values

Syntax

dg

Description

For each data symbol in the debugger's symbol table, **dg** displays the type, name, and value for that symbol. If the symbol is an array of structures, each element in the array will be printed.

- **ds** Displays stack backtrace
- **dS** Displays stack, function arguments, and auto backtrace

Syntax

[#] **ds**
[#] **dS** *command*

Description

The **ds** command displays information about the current function, the function that called it, etc., in reverse order of invocation, back to `__main()`, the Manx function which called the user's function `main()`.

The optional '#' parameter specifies the number of stack frames to be displayed.

For each function, the information consists of the function name and the parameters passed to it.

Arguments are displayed according to their type. If no type information is available, the arguments are displayed as a series of 16-bit hex values. Arguments of type `long` or `double`, when displayed as hex, will be displayed as separate words.

ds determines the number of parameters by looking at the instructions that follow the address to which the function returns.

ds assumes that the `a5` register points to the C stack frame for the current function, unless the current instruction is within 6 bytes of the start of the function.

dS causes the function to be displayed with the types, names, and values of each function argument. Below this, the values of all automatic variables are displayed.

“FIND SOURCE STRING” COMMAND

- / Finds string in source file

Syntax

/*string*

Description

This command searches the current source file (e.g., the one displayed in the source window) for a specified *string*.

The search begins at the line following the current line. If the *string* is found, the current line and the F-dot line of the source file is set to the line containing the *string*; otherwise, these values are unchanged.

string is the character *string* to be located, and consists of all characters following the / and preceding the carriage return.

If the first character of the *string* is ^, the search will begin with the first character on a line. In this case, ^ is not part of the search *string*.

The current line and F-dot line for a source file are defined in the description of the df command.

FRAME COMMANDS

- fu Frame up command
- fd Frame down command

Syntax

fu
fd

Description

Normally, you are only allowed to view variables that are visible by C rules at the point where execution stopped. That is, you cannot refer to local variables of functions that are not the current active function.

sdb allows you to get around this restraint by allowing you to change what the current function is.

The fu command allows you to walk up the call frame and displays the line from which the call was issued. By changing frames, you can view all the local variables of the now current function. You can walk up the frame until you get to the initial function.

The **fd** command allows you to walk down the frame displaying the function call of the previous frame.

GO COMMANDS

- **g** Execute the program
- **G** Execute the program, without setting table breakpoints

Syntax

```
[#]g [addr] [,command]  
[#]G [addr] [,command]
```

Description

The **g** commands transfer control of the processor to your program, at the address specified by the **pc** register. Your program then executes until it terminates, an error (such as division by zero) occurs, or a breakpoint is taken. Control then returns to the debugger program.

The parameters to the **g** commands allow one or two temporary breakpoints to be set in memory before your program is executed.

The difference between the **g** and the **G** command is that the **G** command sets in memory just the breakpoints specified in the command itself, while the **g** command also sets the breakpoints specified in the breakpoint table.

The **#** and **addr** parameters define one of the temporary breakpoints that a **go** command can set:

- **#** is the skip count for the breakpoint. It defaults to zero, meaning that the breakpoint is taken every time it is reached.
- **addr** is the address for the breakpoint.

The **@ <function>** parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function is not specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function returns. In this case, the breakpoint is not set until the function is entered. Thus, in programs

that call the function from several different places, the breakpoint will be set at the actual address to which the function returns.

The *cmdlist* parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to execute once a breakpoint that is specified in the *go* command is taken. If this parameter is not specified, it defaults to the command list used for the last temporary breakpoint.

Before setting breakpoints and transferring control to your program, the debugger single steps your program (that is, causes it to execute one instruction). This allows the operator to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

LOAD COMMANDS

- **ld** load device symbols
- **ll** load library symbols
- **lp** load program

Syntax

```
ld devname
ll libname
lp
lp progfile [arg1 arg2 ...]
```

Description

The **ld** and **ll** commands are used to open a second .dbg file for debugging a library or device. (See the **ax** command as well as the **Debugging Libraries or Device Drivers** section of this chapter for a detailed description of these commands.)

The **lp** command loads a program into memory. If a symbol table file can be found for the program, it is loaded, too.

If the **lp** command is given without parameters, the last **lp** command is reexecuted. The following comments describe the parameterized version of **lp**.

Loading the Program

The parameter *progfile* specifies the file containing the program.

If the *progfile* specifies a drive or path, the file is searched for in just that location; otherwise, it is searched for on the current directory of the default drive.

If an attempt is made to load a program when sdb was invoked with a program name or a previous lp *progfile* was executed, an error will be printed and the command ignored.

Loading the Symbol Table

The name of the file containing the symbol table is assumed to be the same as the program filename, with the extension changed to .dbg.

After the program is loaded, its registers are initialized as though called from the CLI with a pointer to the program's arguments. These are available to the program as the main function arguments *argv[1]*, *argv[2]*, and so on. *argv[0]* is set to the name of the program being debugged, and *argc* is set to the number of *arg* parameters.

The U-DOT (e.g., the value of the period parameter associated with the u commands) is set to Pc. The D-DOT (the value of the period parameter for the d and m commands) is set to 0.

Once a program exits, it must be reloaded with an lp command before it can begin again.

MEMORY MODIFICATION COMMANDS

- mb Modify bytes of memory
- mw Modify words of memory
- ml Modify long words of memory

Syntax

```
mb  addr  val1  [val2 ...]
mw  addr  val1  [val2 ...]
ml  addr  val1  [val2 ...]
```

Description

mb, **mw**, and **ml** modify bytes, words, and long words of memory, respectively. The parameter *addr* specifies the address of the first byte or word to be modified.

The *val* parameters are expressions whose resulting values are set in memory, with *val1* set in the first byte or word specified, *val2* set in the next higher byte or word, and so on.

The *val* parameters can be separated by spaces or commas.

- **mc** Compare memory

Syntax

mc *range* = *addr*

Description

The **mc** command compares two blocks of memory and, for each comparison that fails, displays the corresponding segment, address, and value.

range specifies one of the blocks of memory. The second begins at *addr* and has the same length as the first block.

- **mf** Fill memory with value

Syntax

mf *range* = *val*

Description

The **mf** command sets each byte in a block of memory to a specified value. The *range* parameter specifies the memory block, and *val* an expression whose resulting value is the value to be set in the range.

- **mm** Move memory

Syntax

mm *range* = *addr*

Description

The **mm** command copies one block of memory to another.

The *range* parameter specifies the source block and *addr* the starting address of the block to be modified.

- **ms** Search memory

Syntax

ms *range* = *val1* [*val2* ...]

Description

The **ms** command searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

range specifies the block of memory. The *val* parameters are expressions, each of whose resulting values is one byte of the search sequence.

OUTPUT CONTROL COMMANDS

- **oe** Toggles command echo flag

Syntax

oe

Description

Normally, commands typed into the command area are not displayed in the data area. This command acts as a toggle and causes most commands to be echoed in the data display. This is useful to separate the output of different commands.

- **op** Toggles pause flag

Syntax

op

Description

Normally, when more than one data window full of output is generated, **sdb** prompts

<Hit Any Key to Continue>.

This command acts as a toggle, allowing the output to continue scrolling even though it contains more than one window full of information.

- **ow** Toggles switch to user screen

Syntax

ow {on | off}

Description

Normally, **sdb** only swaps the application screen to the front when you type **g**, **s**, or **t**. This is done to avoid unnecessary screen flicker when single stepping through a program. The only problem is when the application modifies the screen. If this happens, the **sdb** screen is altered unless it has been swapped with your screen. Using this command allows screen swapping to be switched on/off.

PRINT COMMAND

- **p** Generates formatted output

Syntax

P [format] [range] [,next]

Description

The **p** command generates a formatted display of C variables, arrays, and structures, by converting data items in memory to a displayable format directed by its type or the optional conversion string format.

format is an optional list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

addr specifies the address of the first data item that **p** is to convert and display.

In the absence of the format string, p looks at the *addr* to be printed, gets its typing information from the symbol table, builds and executes the appropriate format string. For example, to print a structure named symbol you simply enter:

```
p symbol
```

which might result in:

```
struct symboltb symbol = {  
    int s_flag = 10  
    char *s_name = 0xFF42  
    int s_value[2] = {  
        10, 5  
    }  
}
```

If you then want to display the string pointed to by `symbol.s_name`, you simply type:

```
ps *symbol.s_name
```

The **s** indicates that you are overriding the default format for **pointer** to **char**. The result might be:

```
pointer
```

If you wish to create your own format string, here is a description of the use of format by p: p works its way through the format string, converting and displaying data items in memory as requested by the format string items. When p reaches an item in the format string, it converts the data item at its current address as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed to be ready to process the next data item as directed by the next format string item.

If *addr* is not entered, the starting address is assumed to be the print command's current address.

Normally, this is the address of the first byte beyond the last data item converted by the last p command. However, there is a format item that causes p to remember the address contained in the current data item and then make that the current address after it finishes processing the entire format string.

The format items have the form

[rpt] [indir] [size] desc_code

where

desc_code is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code **d** says “take the two-byte binary value at the current address, convert it to decimal, and print it”. Therefore, if **var** is an **int**, the following command could be used to print its value in decimal:

pd var

The code **x** says “take the two-byte binary value at the current address, convert it to hexadecimal, and print the result”. So the hexadecimal value of **var** could be printed with the command:

px var

A format item’s *indir* is a string that consists of zero or more * characters that are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code*.

To find the data object corresponding to a format item that has indirection indicators, **p** begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address: At the end of this process, the **p** command’s current address is simply incremented past the first pointer.

For example, if the variable **cp** is a short pointer to a character string (e.g., its declaration is **char *cp**), then the string pointed at by **cp** could be printed by the command:

p*s cp

Here we have used the **s** *desc_code*, which specifies that the data object is a character string, and that the string’s characters are to be printed, with possible modifications as noted below, up to a terminating null character.

After this command, the **p** command's current address is set to the byte immediately following **cp**.

The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count. For example, if **a** is an array of floats, then the first five items in this array could be displayed with the command

```
p5f a
```

This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is **f**. This command is equivalent to the command **pfffff a**. At the end of this command, the **p** command's current address is set to the address of the byte following the last displayed float.

The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item does not use indirection, *size* has the same effect as *rpt*; for example, in the **p5f a** command above, the **5** could be interpreted as being a *size* parameter instead of an *rpt* parameter.

When the format item does use indirection, then the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module defines **lpp** as a pointer to an array of pointers to longs (e.g., the declaration of **lpp** is **long **lpp**), then the following command would display the first four longs pointed at by the first element of the pointer array:

```
p**4D lpp
```

Here we have used the *desc_code* **D**, which specifies that a four-byte signed binary value is converted to decimal and printed.

The following command would display the first three longs pointed at by the first three elements of the pointer array:

```
p3*4D *lpp
```

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items **4*d *4d**. The first causes the print command to take the item at the current address as a pointer, increment the current address by two, convert to decimal and print the two-byte value

referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by eight.

The second item causes the print command to again take the item at the current address as a short pointer, increment the current address by two, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has been advanced by two.

As an example of the use of format strings containing several format items, consider the following code:

```
struct {
    int *ip;
    float flt;
    char *cp;
} var = (&i, 3.14159, "ralph");
int i=2;
```

The command:

```
p*d2-xf*s2-x var
```

prints:

```
2 xxxx 3.14159 "ralph" yyyy
```

where **xxxx** is the hexadecimal address of **i** and **yyyy** is the hexadecimal address of the string.

desc_code List

The basic *desc_codes* are as follows:

- b** Converts to hexadecimal and prints a byte.
- d** Converts to decimal and prints a two-byte signed binary value.
- D** Converts to decimal and prints a four-byte signed binary value.
- e** Converts and prints a **long double** floating point value

- f** Converts and prints a double floating point value.
- F** Converts and prints an eight-byte double.
- .** Defines the precision to be used in the display of floating point values, e.g., the number of digits to be displayed to the right of the decimal point. This number precedes the period. For more information, see below.
- o** Converts to octal and prints a two-byte field.
- O** Converts to octal and prints an eight-byte field.
- x** Converts to hexadecimal and prints a 2-byte field.
- X** Converts to hexadecimal and prints a 4-byte field.
- u** Converts to decimal and prints an unsigned, two-byte value.
- U** Converts to decimal and prints an unsigned, four-byte value.
- c** Prints a character.
- C** Prints a character.
- s** Prints a string up to a terminating null byte.
- S** Prints a string with (without) translation.

Floating Point Number Print Codes

When a floating point number is displayed in response to an **f** or **F** *desc_code*, the precision of the display (e.g., the number of digits displayed to the right of the decimal point) is determined by the most recently-entered period *desc_code*. If a period code has not been entered, the precision of the display defaults to 7 digits for floats and 16 for doubles.

The period *desc_code* consists of a period preceded by the number of digits of precision. For example, the following command contains two *desc_codes*. The first sets the precision to 2 digits, the second then displays a float, listing two digits to the right of the decimal point:

```
p2.F
```

Character Print Codes

For the C and S *desc_codes*, each character is printed as is, with no translations. For the c and s codes, printable ASCII characters (e.g., whose hex value is between 0x20 and 0x7f) are printed as is. A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, 0x41, 0x1, and 0x81 would be printed as A, ^A, and '^A, respectively.

Special Purpose Codes

The following *desc_codes* can be used to assist in the formatting of the p output:

Character	Output
N or n	Outputs a newline character
R or r	Prints a name
T or t	Outputs a tab character
"string"	Outputs "string"

These characters can be preceded by a count specifying the number of characters or strings to be output.

P Command *desc_codes* For Setting Current Address

The next group of *desc_codes* change the p command's notion of the current address. They do not cause any printing.

- ^ Backs up the current address by the size of the last data item.
- or + Backs up or advances, respectively, the current address by *size* bytes, where *size* is a decimal value preceding the - code. If *size* is not specified, it defaults to one byte.
- A or a Remembers the long or short pointer, respectively, that is contained in the current data object. If this pointer is not null, sets the p command current address to this value after the entire format string has been processed.

If the pointer is null, sets the p command's current address to the value it had before the entire format string was processed.

The A and a *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines the structure for a symbol table item and declares **sym_head** to be a pointer to this structure. The program that uses this structure and field will chain symbol table items together and set a pointer to the head of the chain in **sym_head**.

```
struct symbol {
    struct symbol * sym_next;
    char *sym_name;
    unsigned sym_val;
    *sym_head;
```

The following command would display the symbol table item pointed at by **sym_head** and then set the p command's current address to the next symbol table item, which is pointed at by the **sym_next** field in the first item:

```
p A"symbol name="#snt"value="x sym_head
```

After this command is entered, you can display successive symbol table items by simply entering

P

The **p** command's current address is correctly set to the next table item and since a format string is not specified, the **p** command uses the one that it last used.

You can print out multiple symbol table items by entering a single **p** command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command follows the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose **sym_next** pointer is null. In the latter case, it will terminate and leave the **p** command's current address set to the address of the last symbol table item. For example, entering

```
p A"symbol name="#snt"value="x sym_head,100
```

prints symbol table items until it either prints 100 items or it prints an item having a null **sym_next** pointer.

If **P** is used, the address of the item is displayed as well.

. QUIT COMMAND

- **q** Quit the debugger

Syntax

```
q
```

Description

sdb terminates the program being debugged by first checking for an **_abort()** code symbol or an **exit()** code symbol. If found, the program counter is set to that symbol and the program is allowed to resume.

This is the normal way to exit **sdb** under most circumstances.

REGISTER COMMAND

- **r** Register display

Syntax

```
r  
r <reg>=val
```

Description

The **r** command displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers. The parameterized version modifies the contents of a register, with *<reg>* being the name of the register to be modified, and *val* an expression whose resulting value is to be set into the register.

If a MC68881 floating point chip is present and not in the reset state, its registers will be displayed as well.

SINGLE STEP COMMANDS

- **s** Single steps the program *n times*
- **S** Single steps into calls without display
- **t** Single steps *n times* stepping over calls
- **T** Single steps over calls, display last

Syntax

```
#s  
#S  
#t  
#T
```

Description

These commands SINGLE STEP your program, e.g., execute its instructions one by one.

The optional # parameter specifies the number of instructions to be executed. It defaults to one instruction.

If the count is zero, it goes forever, e.g., 0T single steps until a breakpoint or an error is encountered.

The s and S commands differ in that s displays information after each single step, whereas S only displays information after the last single step.

The same is true for the t and T commands.

The s and t commands differ in that the s command single step into calls when encountered while the t command treats a function call as a single step and steps over it.

The displayed information consists of the source line of the next instruction to be executed. When single stepping, breakpoints are not enabled.

UNASSEMBLE COMMANDS

- u Unassembles memory, with symbols
- U Unassembles memory, without symbols

Syntax

`u range`
`U range`

Description

These commands DISASSEMBLE a *range* of memory, e.g., display the assembly language instructions in the *range*.

Both the u and U commands make use of the symbol table during disassembly. They differ in that the u command includes the symbols+offset as part of the assembly language while the U command prints locations in hex with the symbol and offset for the location displayed at the far right of the line. Also, the U command displays, for each instruction, the hex value of each byte of the instruction, whereas the u command will not.

With the u command, the disassembly of an instruction that references memory displays the location as the symbol nearest to the location plus an

offset, if possible. With the U command, the location is displayed as a hexadecimal value.

The *range* parameter specifies the area of memory to be disassembled. It gives the starting address and either the number of instructions to be disassembled or the ending address of the area.

Only the u (lowercase) command displays the source lines and autos as comments.

MACRO COMMANDS

- x Create Macro Command
- X Display Macro Command

Syntax

x name macro
X name

Description

Macro The x command defines or executes a sequence of debugger commands, called a macro. X lists the defined macros.

A *macro name* consists of a sequence of alphanumeric characters, the first of which must be alphabetic, with a limit of 40 characters. Case is not significant.

A macro is defined by typing the letter x, followed by the *name* with which the macro is to be associated. Then follows the macro's list of debugger commands with the commands separated by semicolons.

A macro is executed by typing x, followed by the *name* with which the macro is associated, followed by a carriage return.

The macros that have been defined can be listed using the command X.

Macros are automatically saved in a file called *xxx.mac* where *xxx* is the same as the name of the program being debugged. This file is executed automatically when sdb is started.

EXPRESSION COMMANDS

- = Display the value of an expression in several formats
- e Evaluate an expression

Syntax

= *expr*
e *expr*

Description

The = command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a symbol table has been loaded, the closest symbol is displayed as well.

The e command allows you to perform C expression evaluation, including calls to C functions, assignment, pre- and post- increment and decrement, casts, and conditionals. For example:

e c = getchar()

might result in:

=10

REDIRECT COMMAND INPUT/OUTPUT

- < Redirect command input
- > Log all I/O only
- >> Log commands only

Syntax

<*filename*
>*filename*
>>*filename*

Description

The < command causes the debugger to redirect input from a file. This command provides a convenient means for defining macros or variables.

When the end of the file is reached, the debugger returns to the console for commands.

The > command causes the debugger to copy all output to the specified file. This feature allows you to keep a record of the commands and responses for a debugging session.

The >> command causes the debugger to log commands only to the specified file. This is useful if you wish to reexecute a sequence of commands to reproduce a problem.

HELP COMMAND

- ? List command

Syntax

?

Description

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the letter, the ?, and return.

For example, the listing for the display commands is d?. Thus you can type d? followed by return to get a listing of all the display commands.

CHANGE MODE COMMAND

- z Change mode command

Syntax

z

Description

The **z** command allows you to switch from source mode to assembly mode and back again. By default, the debugger starts in source mode unless you specify option **-a** on the command line.

DELAY COMMAND

► & Delay sdb

Syntax

& [*cnt*]

Description

This command delays **sdb** for *cnt* seconds or until you hit return if a *cnt* is not given. Its primary purpose is for use in the demo scripts where the application screen is displayed for a couple of seconds while **sdb** waits.

Command Summary

AMIGA SPECIFIC COMMANDS

acc	show/set the command window colors
acd	show/set the data window colors
acs	show/set the source window colors
add	display device list
adi	display interrupt list
adl	display library list
adp	display port list
adr	display resource list
am	display memory usage
ax	switch between main and alternate .dbg file

BREAKPOINT COMMANDS

bc/bC	clear one/all breakpoints
bd	display the breakpoint table
be	set expression breakpoint
bm	set memory change breakpoint
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode

DISPLAY COMMANDS

c	display source context
da/dA	display local variables/addresses
db/dw/dl/d	display memory in bytes/words/long words/last format
dc/dd	display code/data symbols
df	display source file lines
dg	display global values
ds/dS	display stack backtrace

FIND SOURCE STRING COMMAND

/ find string in source file

CHANGE FRAME COMMANDS

fu walk up the call frame
fd walk down the call frame

GO COMMAND

g/G execute program

LOAD COMMAND

ld load device symbols
ll load library symbols
lp load program

MEMORY MODIFICATION COMMANDS

mb/mw/ml modify bytes/words/long words of memory
mc compare areas of memory
mf fill memory
mm move memory
ms search memory

OUTPUT CONTROL COMMANDS

oe toggle command echo flag
op toggle pause flag
ow toggle switch to user screen

FORMATTED PRINT COMMAND

p generate formatted print

QUIT COMMAND

q quit debugger

REGISTER COMMAND

r register display

SINGLE STEP COMMANDS

s/S single step with/without display, stepping into function calls
t/T single step with/without display, stepping over function calls

UNASSEMBLY COMMANDS

u/U unassemble memory

MACRO COMMANDS

x/X define or execute/display a command macro

EXPRESSION COMMANDS

= display value of an expression in several formats
e evaluate an expression

REDIRECT COMMAND INPUT

< redirect command input
> log all input/output
>> log commands only

HELP COMMAND

? list debugger commands

CHANGE MODE COMMAND

z switch from source to assembly mode and back

DELAY COMMAND

& delay until return or specified seconds

Index

#acs command, 4-46
\$ character, 4-8
& command, 4-45, 4-48
-a option, 2-4
-cc option, 2-4
-cd option, 2-4
-cs option, 2-4
-cw option, 2-5
-d option, 2-5
-e option, 2-5
-h option, 2-5
-m option, 2-5
-n option, 2-5
-p option
 -w mode, 2-5
-r option, 2-5
 use with REZ utility, 2-5
-s option, 2-5
-w option, 2-5
/ command, 4-7, 4-25, 4-47
68010, 1-2
68020, 1-2
68030, 1-2
< command, 4-11, 4-44
 See also redirect commands
= command, 4-11, 4-43, 4-48
 expression commands, 4-43
> command, 4-11, 4-44
 See also redirect commands
>> command, 4-11, 4-44
 See also redirect commands
? command, 4-2, 4-11, 4-44, 4-48

A

acc command, 4-15, 4-46
 description, 4-15
acd command, 4-15, 4-46
acs command, 4-15 - 4-16
add command, 4-4, 4-16, 4-46
ADDR, 4-14
 definition, 4-13
 examples, 4-13
adi command, 4-16, 4-46
adl command, 4-4, 4-16, 4-46
adp command, 4-16, 4-46
adr command, 4-16, 4-46
am command, 4-17, 4-46
Amiga specific commands
 listing, 4-46
 acc command, 4-15
 acd command, 4-15
 acs command, 4-15 - 4-16
 add command, 4-16
 adi command, 4-16
 adl command, 4-16
 adp command, 4-16
 adr command, 4-16
 am command, 4-17
 ax command, 4-17
 ax command, 4-4, 4-17, 4-46

B

b commands, 4-5
backtracing, 4-6
basic commands
 display, 4-1
 evaluate, 4-2
 frame, 4-2
 help, 4-2
 memory modify, 4-2
 print, 4-2
 register, 4-2

single step, 4-2
transfer control commands, 4-2
unassemble, 4-2
bc/bC command, 4-17, 4-46
bd command, 4-18, 4-46
be command, 4-6, 4-18, 4-46
bin directory, 2-2
bm command, 4-19, 4-46
br command, 4-20, 4-46
breakpoint commands, 4-5
 See also specific command
b, 4-17 - 4-18
bc/bC, 4-17
bd, 4-18
be, 4-18
bm, 4-19
br, 4-20
bs, 3-4, 4-20
bt/bT, 4-20
listing, 4-46
breakpoints, 4-5 - 4-6
 expression, 4-18
 memory change, 4-19
 one-time, 3-3
 permanent, 3-3 - 3-4
bs command, 4-20, 4-46
 examples, 3-4
bt/bT command, 4-6, 4-20, 4-46

C

c command, 3-2, 4-7, 4-21, 4-46
C source file, 2-3
call trace mode, 4-20
 enabling and disabling, 4-6
categories
 backtracing, 4-6
 basic commands, 4-1 - 4-2
 breakpoints, 4-5 - 4-6
 command display, 4-9
 command line, 4-10

command line editing, 4-9 - 4-10
debugging device drivers, 4-4
debugging libraries, 4-4
displaying source files, 4-7
expression breakpoints, 4-6
loading programs and symbols, 4-3
macros, 4-7
names, 4-3
other features, 4-11 - 4-12
source and data displays, 4-8
term definitions, 4-12 - 4-14
trace mode, 4-6
windows, 4-7 - 4-8

cd option, 2-4

change frame commands
listing, 4-47

change mode command
See z command
listing, 4-48
z, 4-45

character print codes, 4-37

CLI, 2-2

CMDLIST
definition, 4-14

code and data symbols, 4-3

color control, 4-10
window colors, 2-5

command description
general, 4-1

command display, 4-9

command input
redirection, 4-11

command line editing, 4-9 - 4-10
^C, 4-10
^X, 4-10
ESC character, 4-10
F10 function key, 4-10

command summary
amiga specific commands, 4-46
breakpoint commands, 4-46

change frame commands, 4-47
change mode command, 4-48
delay command, 4-48
display commands, 4-46
expression commands, 4-48
find source string command, 4-47
formatted print command, 4-47
go command, 4-47
help command, 4-48
load command, 4-47
macintosh specific commands, 4-46
macro command, 4-48
memory modification commands, 4-47
output control commands, 4-47
quit command, 4-47
redirect command input, 4-48
register command, 4-48
single step commands, 4-48
unassembly commands, 4-48
commands, detailed descriptions
 See also SDB commands
 Amiga specific, 4-14
 breakpoint, 4-17 - 4-19
 change mode, 4-45
 delay, 4-45
 display, 4-21 - 4-24
 expression, 4-19, 4-43
 find source string, 4-25
 frame, 4-25
 go, 4-26
 help, 4-44
 load, 4-27
 macro, 4-42
 memory modification, 4-28 - 4-29
 output control, 4-30
 print, 4-31 - 4-36, 4-38 - 4-39
 quit, 4-39
 redirect input/output, 4-44
 register, 4-40
 single step, 4-40 - 4-41

unassemble, 4-41 - 4-42
compare memory command, 4-29
compiler
 -bs option, 2-3, 4-4
control characters
 ^, 4-12
 ^C, 4-10
 ^D, 4-12
 ^G, 4-12
 ^K, 4-12
 ^L, 4-12
 ^N, 4-12
 ^T, 4-12
 ^U, 4-12
 ^V, 4-12
 ^W, 4-12
 ^X, 4-10
 ^Y, 4-12
control characters, use of, 4-12
count, 3-2
loading sdb
 creating "work disk", 2-2
cs option, 2-4
current source line, 3-2
cw option, 2-5

D

d command, 4-5, 4-21, 4-46
da/dA command, 4-21, 4-46
db command, 4-21, 4-46
dbg file command
 switch between main and alternate, 4-5
dc command, 4-22, 4-46
dd command, 4-22, 4-46
debug file, 2-3
debugging device drivers, 4-4
debugging libraries, 4-4
default settings, 4-11
delay command
 &, 4-45

listing, 4-48
demonstration programs, 2-1, 2-3
desc_code list, 4-35 - 4-38
desc_codes
 character print codes, 4-37
 examples, 4-33 - 4-34, 4-38
 floating point number print codes, 4-36
device drivers
 debugging, 4-4
df command, 3-2, 4-7, 4-22 - 4-23, 4-46
 range, 3-2
dg command, 4-23, 4-46
disassembly
 See unassemble commands
display code command, 4-3
display code symbols, 4-22
display command, 4-5
display commands, 4-2
 See also specific commands
=, 4-43
add, 4-16
adi, 4-16
adl, 4-16
adp, 4-16
adr, 4-16
am, 4-17
bd, 4-18
c, 4-21
d, 4-21
da/dA, 4-21
db, 4-21
dc, 4-22
dd, 4-22
df, 4-22 - 4-23
dg, 4-23
dl, 4-21
ds/dS, 4-24
dw, 4-21
listing, 4-46
r, 4-40

s/S, 4-40
t/T, 4-40
x/X, 4-42
display context command, 4-21
display data symbols, 4-22
display global value, 4-23
display memory commands, 4-21
display register command, 4-40
display source file lines, 4-22
display symbols command, 4-3
displaying assembly, 3-7
displaying information
 current function, 4-7
displaying source files, 3-2, 4-7
displaying the trace of calls, 3-5
displaying values and computing expressions, 3-6
displaying variables, 3-6
dl command, 4-21, 4-46
ds/dS command, 3-5, 4-7, 4-24, 4-46
 example, 3-5
dw command, 4-21, 4-46

E

e command, 3-6, 4-2, 4-43, 4-48
 example, 3-6
environment variables
 SDBOPT, 2-5, 4-11
evaluate command, 3-6, 4-2
evaluate expression command, 4-12
execute program commands, 4-26
execution
 controlling, 3-3
EXPR
 definition, 4-13
 example, 4-13
expression breakpoints, 4-6
expression commands
 =, 4-43
 e, 4-43
 listing, 4-48

F

f command, 4-2
fd command, 3-7, 4-25, 4-47
file preparation
 -bs compiler option, 2-3
 -g linker option, 2-3
 executable files, 2-3
files for debugging
 preparation of, 2-1
find source string commands, 4-7
 /, 4-25
 listing, 4-47
floating point number print codes, 4-36
format items
 desc_codes, 4-34 - 4-35
 examples, 4-34 - 4-35
formatted print command
 listing, 4-47
frame commands, 4-2
 fd, 3-7, 4-25
 fu, 3-7, 4-25
fu command, 3-7, 4-25, 4-47

G

g/G command, 3-4, 4-2, 4-5, 4-26, 4-47
go command, 3-4
 See also g/G command
controlling execution, 3-3
g/G, 4-26
listing, 4-47

H

help command, 4-2, 4-12
 ?, 4-44
 listing, 4-48
help screens, 3-1

I

- installation, 2-2
 - getting started, 2-1
 - using floppy disk, 2-2
 - using hard disk, 2-2

K

- keys
 - See also control characters

L

- ld command, 4-4, 4-27, 4-47
 - line, 3-2
 - linker
 - g option, 2-3, 4-3 - 4-4
 - list command
 - See help command
 - ll command, 4-4, 4-27, 4-47
 - load command
 - description, 4-27
 - ld, 4-27
 - listing, 4-47
 - ll, 4-27
 - loading the program, 4-28
 - loading the symbol table, 4-28
 - lp, 4-27 - 4-28
 - load device symbols command, 4-27
 - load library symbols command, 4-27
 - load program command, 4-3, 4-27
 - loading programs and symbols, 4-3
 - loading sdb
 - using floppy drives, 2-2
 - using hard disk, 2-2
- log commands, 4-11
 - <, 4-43
 - >, 4-43
 - >>, 4-43
- lp command, 2-4, 4-3, 4-27 - 4-28, 4-47

M

macro commands
 listing, 4-48
 x/X, 4-42

macros
 defining and executing, 4-7
 fpath, 2-6

mb command, 4-28, 4-47

mc command, 4-29, 4-47

memory modification commands, 4-2
 listing, 4-47
 mb, 4-28
 mc, 4-29
 mf, 4-29
 ml, 4-28
 mm, 4-30
 ms, 4-30
 mw, 4-28

mf command, 4-29, 4-47

ml command, 4-28, 4-47

mm command, 4-30, 4-47

mode commands
 See change mode command

move memory command, 4-29

ms command, 4-30, 4-47

mw command, 4-28, 4-47

N

names
 code and data symbols, 4-3
 operator usage of, 4-3

notation conventions, 1-2

O

oe command, 4-30, 4-47

op command, 4-31, 4-47

operator usage of names, 4-3

options
 -a , 2-4

- cc, 2-4
- cd , 2-4
- cs , 2-4
- cw , 2-5
- d , 2-5
- e , 2-5
- h , 2-5
- m, 2-5
- n , 2-5
- p , 2-5
- r , 2-5
- s, 2-5
- w , 2-5
- options list, 2-5
- other features
 - control characters, use of, 4-12
 - evaluate expression command, 4-11
 - help command, 4-11
 - redirect input command, 4-11
- output commands
 - listing, 4-47
- output control commands
 - oe, 4-30
 - op, 4-31
 - ow, 4-31
- ow command, 4-31, 4-47

P

- p command, 4-2, 4-31 - 4-39, 4-47
 - current address setting codes, 4-38 - 4-39
 - example, 3-6
- parameters, 2-5
- Preferences program, 4-10
- preparation of files for debugging, 2-1, 2-3
- print command, 3-6, 4-2
 - character print codes, 4-37
 - current address setting codes, 4-38 - 4-39
 - desc_code list, 4-35 - 4-38
 - description, 4-31 - 4-34
 - examples, 4-32 - 4-34

floating point number print codes, 4-36
format items, 4-33, 4-35
p, 4-31 - 4-39
special purpose codes, 4-37
processors supported, 1-2
program
 running, 3-3 - 3-4

Q

q command, 4-39, 4-47
quit command
 listing, 4-47
 q, 4-39

R

r command, 4-2, 4-40, 4-48
range, 3-2, 4-14
 definition, 4-13
installation, 2-1
README file, 1-2, 2-1
redirect command input
 listing, 4-48
redirect commands, 4-44
 <, 4-43
 description, 4-44
redirection, 4-11
register command, 4-2
 listing, 4-48
 r, 4-40
requirements, 1-2
running the program, 3-3 - 3-4

S

s option
 fpath macro, 2-6
s/S command, 3-3, 4-2, 4-40 - 4-41, 4-48
scratch directory, 2-2
SDB
 definition, 1-1

SDB commands, 3-7

See also specific command
&, 4-45
/, 4-7, 4-24
<, 4-11, 4-43
=, 4-43
>, 4-11, 4-43
>>, 4-11
?, 4-44
ac, 4-11
acc, 4-15
acd, 4-15
acs, 4-15
add, 4-4, 4-16
adi, 4-16
adl, 4-4, 4-16
adp, 4-16
adr, 4-16
am, 4-17
ax, 4-17
b, 4-5
bc/bC, 4-17
bd, 4-18
be, 4-6, 4-18
bm, 4-19
br, 4-19
bs, 3-4, 4-20
bt/bT, 4-6, 4-20
c, 3-2, 4-7, 4-21
d, 4-5, 4-21
da/dA, 4-21
db, 4-1, 4-21
dc, 4-3, 4-22
dd, 4-3, 4-22
df, 3-2, 4-7, 4-22 - 4-23
dg, 4-23
dl, 4-1, 4-21
ds/dS, 3-5, 4-6, 4-24
dw, 4-1, 4-21
e, 3-6, 4-2, 4-43

f, 4-2
fd, 3-7, 4-25
fu, 4-25
g/G, 3-4, 4-2, 4-6, 4-26 - 4-27
h, 4-2
ld, 4-4, 4-27
ll, 4-4, 4-27
lp, 2-6, 4-3, 4-27 - 4-28
m, 4-2
mb, 4-28
mc, 4-29
mf, 4-29
ml, 4-28
mm, 4-29
ms, 4-30
mw, 4-28
oe, 4-30
op, 4-30 - 4-31
ow, 4-31
p, 4-1 - 4-2, 4-31 - 4-39
p , 3-6
q, 4-39
r, 4-2, 4-40
s/S, 4-2, 4-6, 4-40
t/T, 3-3, 4-2, 4-40
u/U, 3-7, 4-2, 4-41
x/X, 4-7, 4-42
z, 4-44
SDBOPT environment variable, 2-5, 4-11
search memory command, 4-30
single step command
 s/S , 3-3
single step commands, 4-6
 controlling execution, 3-3
 default settings, 3-3
 listing, 4-48
 s/S, 4-40 - 4-41
 t/T, 4-40 - 4-41
sizing icon, 4-7
source and data displays, 4-8

source file, displaying sections, 3-2
source line trace mode
 enabling and disabling, 4-6
source window, 3-2
special purpose codes, 4-37
startup, 2-6
startup SDB, 2-1, 2-4
 from Shell or CLI, 2-6
 from the Aztec Shell, 2-4
statics, 3-7

T

t/T command, 3-3, 4-2, 4-40 - 4-41, 4-48
technical support, 1-3
term definitions
 ADDR, 4-13
 CMDLIST, 4-14
 EXPR, 4-12
 RANGE, 4-13
trace mode, 4-6
trace mode commands, 4-6
 bt/bT, 4-20
transfer control commands, 4-2, 4-5
truncated lines, 4-8
tutorial
 introduction, 3-1 - 3-2

U

u/U command, 3-7, 4-41 - 4-42, 4-48
 See also unassemble commands
unassemble commands, 4-2
 disassembly, 3-7
 listing, 4-48
u/U, 4-41 - 4-42

W

walking up and down the frames, 3-7
windows, 4-7
 color control, 2-5

colors, 2-5, 4-10
source and data display, 4-8
Workbench colors, 4-10

X

x/X command, 4-7, 4-42, 4-48

Z

z command, 3-3, 4-45, 4-48

Manx Software Systems, Inc.