Aztec C65
for the Apple //

version 3.2

July 1986

Distributed by:
Manx Software Systems, Inc.
P.O. Box 55
Shrewsbury, N.J. 07701
201-542-2121

USE RESTRICTIONS

The components of the Aztec C65 software development system are licensed software products. Manx Software Systems reserves all distribution rights to these products. Use of these products is prohibited without a valid license agreement. The license agreement is provided with each package. Before using any of these products the license agreement must be signed and mailed to:

Manx Software Systems
P. O. Box 55
Shrewsbury, N. J 07701

The license agreement limits use of these products to one machine. Any uses of these products that might lead to the creation of or distribution of unauthorized copies of these products will be a breach of the licensing agreement and Manx Software Systems will excercise its right to reclaim the original and any and all copies derived in whole or in part from first or later generations and to pursue any appropriate legal actions.

Software that is developed with Aztec C65 software development system can be run on machines that are not licensed for these products as long as no part of the Aztec C software, libraries, supporting files, or documentation is distributed with or required by the software. In the latter case a licensed copy of the appropriate Aztec C software is required for each machine utilizing the software. There is no licensing required for executable modules that include runtime library routines.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. DAC #84-1, 1 March 1984. DOD Far Supplement.

COPYRIGHT

DISCLAIMER

TRADEMARKS

Manual Revision History

Nov 1985 .... First Edition
July 1986 .... Second Edition

Summary of Contents

Contents

OVERVIEW

Overview

The Aztec C65 Software Development Package is a set of programs for developing programs in the C programming language; the resulting programs run on an Apple // that use either ProDOS or DOS 3.3. The development is done using ProDOS.

Some of the features of Aztec C65 are:

* The full C language, as defined in the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, is supported.
* Development is done using a program called the SHELL, which provides a UNIX-like environment.
* You can easily define to the SHELL and other programs the devices that are on your system.
* C programs can be translated into native 6502 or 65C02 code, or into "pseudo code". A program's native code is directly executed by the processor, while its pseudo code is executed by an Aztec routine that is in the program. A program can contain both native and pseudo code.
* A full-screen editor, *ved*, is provided; With some versions of Aztec C65, utility programs are provided that are similar to UNIX programs: *grep*, a pattern-matcher; *diff*, a program that determines the difference in source files;
* An extensive set of user-callable functions is provided.
* Features and functions are provided that allow programs to call the operating system.
* Code can be partitioned into overlays, allowing programs to be created and executed that are larger than available memory.
* Modular programming is supported, allowing the components of a program to be compiled separately, and then linked together.
* Programs can be developed that can only be activated from within the SHELL environment. Such programs have many UNIX features.
* Programs can also be developed that can be activated from within either the SHELL or Basic Interpreter environments. Such programs have fewer UNIX features than do those that

can only be executed within the SHELL environment.

* Assembly language code can either be combined in-line with C source code, or placed in separate modules which are then linked with C modules.

There are two classes of user-callable functions: system independent and system dependent. The system-independent functions are compatible with their UNIX counterparts and with the system-independent functions provided with Aztec C packages for other systems. Use of these functions allows programs to be recompiled for use on UNIX-based systems or on other systems supported by Aztec C with little or no change.

The system-dependent functions allow programs to take advantage of special features of the Apple //.

Versions

Several versions of the Aztec C65 Software Development System are available, for use in different environments. Some, called "native development systems", allow development to be done on the Apple // running ProDOS; the others, called "cross development systems", allow development to be done on other machines, with the resulting programs downloaded to the Apple //.

For information about the systems on which cross development can be done, see the Aztec C65 product bulletins.

Requirements

To develop Apple // programs on an Apple using Aztec C65, your Apple must meet the following requirements:

* It must be able to run ProDOS.
* It must have at least two floppy disk drives.
* It must support keyboard entry of lower case characters.
* It must support console display of the full set of displayable ASCII characters.

If you have an Apple // or Apple // Plus, support for the keyboard and console requirements can be provided by installing the single wire shift key mod and by installing a lower case adaptor. For information, see your Apple dealer.

Components

Aztec C65 contains the following components:

* *cc* and *as*, the native-code compiler and assembler;
* *cci* and *asi*, the interpretive-code compiler and assembler;
* *ln*, the linker;

* *lb*, the object module librarian;
* The SHELL, a command processor that replaces the Basic Interpreter and provides a UNIX-like environment in which to develop programs;
* Object libraries containing user-callable functions and support functions;
* Several utility programs, including, with some versions of Aztec C65, programs similar in function to the UNIX utilities *grep* and *diff*.

Preview

This manual is divided into two sections, each of which is in turn divided into chapters. The first section presents Apple-specific information; the second describes features that are common to all Aztec C packages. Each chapter is identified by a symbol.

The Apple-specific chapters and their identifying codes are:

tutor describes how to get started with Aztec C65: it discusses the installation of Aztec C65, presents an introduction to the SHELL, and gives an overview of the process for turning a C source program into an executable form;

sh describes the SHELL;

cc, *as*, and *ln* present detailed information on the compilers, assemblers, and linker;

util describes the utility programs that are provided with Aztec C65;

libov65 describes Apple-specific overview information;

lib65 describes the special, Apple-specific functions provided with Aztec C65;

tech discusses several miscellaneous topics, including memory organization, creation of command programs, overlays, writing assembly language functions, and debugging;

The System-independent chapters and their codes are:

libov presents an overview of the system-independent features of the functions provided with Aztec C65;

lib describes the system-independent functions provided with Aztec C65;

style discusses several topics related to the development of C programs;

err lists and describes the error messages which are generated by the compiler and linker.

TUTORIAL INTRODUCTION

Chapter Contents

Tutorial Introduction

This chapter describes how to quickly start using Aztec C65 for ProDOS.

We first present the steps to prepare a set of disks with which to develop programs. Then we describe the steps to make an executable version of a sample C program whose source is on one of the Aztec C65 disks. Then we introduce some of the features of Aztec C65, including, in particular, the SHELL. This is the Aztec C65 command processor program, which replaces the ProDOS Basic Interpreter program and provides a UNIX-like environment in which to develop programs. Finally, we introduce the rest of the manual.

1. Prepare Disks for Development

To get started developing programs with Aztec C65, you need a copy of the Aztec C65 distribution disks and a "working disk" on which you'll place your own files. In this section we'll lead you through the steps to create these disks, using the ProDOS *filer* program. Once you've made a copy of the distribution disks, place the originals in a safe place.

1.1 Start the *filer*

To start the *filer*, follow these steps:

1. Put the Aztec C65 disk labeled */system* in drive 1 and turn on the Apple. This will start ProDOS, loading it into memory from the file named *prodos* that's on the */system* disk, and then transferring control of the processor to it. ProDOS will then start the Aztec program named *SHELL*, loading it from the file named *shell.system* on the */system* disk, and then transferring control of the processor to it. The *SHELL* is the program to which you will enter commands while developing programs. The SHELL will display the prompt -?, indicating that it is ready for you to enter a command.
2. Enter the command

   ```
   filer
   ```

   and then type the *Return* key. The SHELL will start the *filer*, loading it into memory from the file on the */system* disk named *filer* and transfering control of the processor to it. The *filer* will display its main menu and wait for you to enter a command.

1.2 Make a Copy of the Distribution Disks

With the *filer* running, follow these steps to make a copy of the Aztec distribution disks:

1. Place one of the distribution disks and a blank disk in the disks drives.
2. With the *filer*'s main menu displayed, type *V*. This will bring up the *filer*'s menu of volume-related commands.
3. With the *filer*'s menu of volume-related commands displayed, type *C*. This will bring up the *filer*'s disk-copying screen.
4. Enter the slot and drive numbers of the drives that contain the distribution disk and the blank disk, and then type the Return key.
5. The *filer* will set the name of the blank disk to that of the distribution disk, and wait for you to select a different name.
6. Don't change the name that the *filer* has selected for the copy of the distribution disk; just type the *Return* key.
7. The *filer* will copy the contents of the distribution disk onto the blank disk, after formatting the blank disk.
8. When the copy is completed, the *filer* will then allow you to copy another disk. Continue until you have copied all the distribution disks.

1.3 Make a working disk

After the last distribution disk is copied, type the *ESC* key to exit the copy command and return to the menu of volume-related commands.

You next need to format a "working disk", which is a disk onto which you will place your own files. To do this, with the menu of volume-related commands displayed type *F*. The *filer* will display a screen for the volume command. Place a blank disk in a drive and enter the slot and drive numbers of the drive to the *filer*, select a name for the disk, and type the *Return* key.

1.4 Exit the *filer* and Restart the SHELL

You have now prepared all the disks that you'll need to start developing programs, so it's time to exit the *filer* and restart the *SHELL*.

The *filer* still has its format screen displayed, and is waiting for your approval to format another disk. You only need one working disk right now, so type the *ESC* key to return to the menu of volume-related commands. Type *ESC* again to return to the *filer*'s main menu. Put your copy of the */system* disk back in drive 1 and type *Q*. The *filer* will display a screen for the Quit command, and wait for you to enter the name of the command processor program that you want it to start. By default, the *filer* assumes that this is the Basic interpreter, in the file *basic.system* on the */system* disk. You want to use the *SHELL* as the command processor, so type the name of the file that contains it, *shell.system*, over the name *basic.system*. Now type the *Return* key.

The *filer* will load the *SHELL* into memory and tranfer control of the processor to it. The *SHELL* will display its *-?* prompt, indicating that it's ready for you to enter a command.

1.5 Copy the SHELL to the Working Disk

The last step necessary to prepare disks for developing programs using Aztec C65 is to copy *shell.system* to the working disk and then restart the SHELL from this new file.

For example, assuming that your working disk is named */work*, the first of the following commands copies *shell.system* to the working disk; the second command reloads the SHELL from the working disk:

```
cp shell.system /work/shell.system
/work/shell.system
```

If your system has a ram disk, you could alternatively copy *shell.system* to the ram disk and reload it from there. The commands would look like this:

```
cp shell.system /ram/shell.system
/ram/shell.system
```

Later in this tutorial we will talk more about the reasons for copying and reloading the SHELL.

1.6 Copy *exmpl.c* to the Working disk

In the next section, we are going to lead you through the steps to convert a C source program to executable form. The source for this program is in the file *exmpl.c* on the */system* disk. You'll need to have this file on your working disk, so, again assuming that this disk is named */work*, enter the following command to copy *exmpl.c* to the working disk:

```
cp exmpl.c /work/exmpl.c
```

2. Creating an executable program

As promised, in this section we will lead you through the steps necessary to translate the sample C program whose source is in *exmpl.c* into an executable form, and then execute it.

For these steps (and later on, for the development of your own programs), you will have your working disk in drive 2 and will swap disks containing the Aztec programs in and out of drive 1.

2.1 Get Ready

If the SHELL isn't already active from the disk preparation that you did previously, start it in the usual way: put the */system* disk in drive 1; turn on the Apple; and wait for the *SHELL* to display its *-?* prompt.

Then put the working disk in drive 2 and enter the command

```
/work/shell.system
```

to make the *shell.system* file that is on the working disk the copy from which the SHELL will be loaded following the execution of a command.

Next, enter the command

```
cd /work
```

to set the "current directory" to the volume directory on the */work* disk.

When a command doesn't explicitly specify the directory that contains a file, it's assumed to be in the "current directory". Since the files that you will create in the following steps will be in the volume directory of the working disk, it simplifies the entry of commands to make this directory the current directory.

2.2 Create the Source Program

The first step to creating a C program is, of course, to create a disk file containing its source. This step isn't needed for this demonstration, since the source code already exists in the file *exmpl.c* that is supplied with Aztec C65. To list the contents of this file, enter

```
cat exmpl.c
```

For your own programs, you can create the C source using any text editor, including the *ved* editor that is provided with Aztec C65. For information on *ved*, including a tutorial introduction to it, see the Utility Programs chapter.

2.3 Compile and Assemble

Put the copy of the */cc* distribution disk in drive 1 and enter the command:

```
cc exmpl.c
```

This compiles the C source that's in *exmpl.c*, translating the C source code into assembly language source and writing it to a temporary file. When done, *cc* starts the *as* assembler. *as* assembles the assembly language source for the sample program, translating it into object code and writing the object code to the file *exmpl.o* in the current directory. When done, *as* deletes the temporary file, which is no longer needed.

Aztec C65 contains two compilers and two assemblers with which you can develop programs. As you've seen, one compiler and its associated assembler are on the */cc* disk. The other compiler and assembler, *cci* and *asi*, are on the */cci* disk. We'll discuss the differences in these compilers and assemblers below.

2.4 Link

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

To do this, put the Aztec disk labeled *ln* in drive 1 and enter the command:

```
ln exmpl.o /ln/c.lib
```

The output of the *ln* linker is sent to the file *exmpl* in the current directory.

During the link step, the linker will search libraries specified to it; when it finds a module containing a needed function, it will include the module in the executable file it's building.

All C programs need to be linked with *c.lib.* (or an equivalent, as discussed below). This library contains the non-floating point functions which are defined in the functions chapter, *lib* of this manual. It also contains functions which are called by compiler-generated code.

If a program performs floating point operations, it must also be linked with a math library. The math library that you will use when getting familiar with Aztec C is *m.lib.* Another version can also be used, as described below.

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

```
ln exmpl.o /ln/m.lib /ln/c.lib
```

2.5 Execute

To execute *exmpl,* enter the name of the file that contains it:

```
exmpl
```

2.6 Cleanup

You now have several files on your working disk that are related to the sample program: *exmpl.c*, *exmpl.o*, and *exmpl.* You may want to keep *exmpl.c* as an example; you don't need *exmpl.o* or *exmpl,* so remove them by entering

```
rm exmpl.o exmpl
```

3. More on the SHELL

Now that you're gone through the creation and execution of a C program, we want to introduce you to the commands that let you examine and move around in a ProDOS file system. For this, we'll assume that you've just finished creating and executing the exmpl program; thus, the */work* working disk is in drive 2, the */ln* disk is in drive 1, and the current directory is the volume directory of the */work* disk. We'll also assume that you have a ram disk named */ram.*

3.1 The *ls* Command

The *ls* command displays information about files and directories. To display the contents of the current directory, enter

```
ls
```

To display the contents of the volume directory on the */ln* volume (by convention, a disk has the same name as its volume directory), enter:

```
ls /ln
```

If you ask *ls* to display information about files or directories that aren't on mounted disk drives, *ls* will display an error message. For example, since */cc* isn't in a drive you would get this message if you entered the command

```
ls /cc
```

You would also get this message if you remove the disk that contains the current directory and enter the command

```
ls
```

To display the names of the volume directories of the disks that are in drives, enter

```
ls /
```

A directory can contain entries to other directories. In an *ls* list, names of directories that are in a directory are preceded by a dash, '-'.

3.2 The Current Directory

The "current directory" is the directory in which you are most interested at a given moment: when you enter a command, the SHELL assumes, unless you specify otherwise, that the command is to access the current directory. That's why typing *ls* without any parameters caused *ls* to display the contents of the current directory: you didn't specify otherwise, so *ls* assumed that you were interested in the current directory. To see what the current directory is, enter the command

```
pwd
```

The *cd* command changes the current directory. For example, entering

```
cd /ln
```

makes the volume directory on the */ln* disk the current directory. If you now reenter the *ls* command without any parameters, it will display the files that are in the */ln* directory.

If the directory specified in the *cd* command isn't on a mounted disk, the current directory will remain unchanged.

3.3 File Names

A file in the current directory can be identified by just specifying its name; for example, with the */work* directory as the current directory, the contents of the file *exmpl.c* that is in this directory can be displayed by entering

```
cat exmpl.c
```

To completely identify a file, you must precede its name with the path of directories that must be passed through to get to it. If you don't specify this path, the file is assumed to be in the current directory. The above *cat* command didn't explicitly say where *exmpl.c* was, so it was assumed to be in the current directory, */work*. An equivalent command, that explicitly defines the directory that contains *exmpl.c*, is

```
cat /work/exmpl.c
```

4. Types of Commands

The SHELL can execute three types of commands: builtins, programs, and exec files:

* The code for a builtin command is contained in the SHELL. *ls*, *pwd*, and *cd* are some of the SHELL's builtin commands. To execute a builtin command's code, the SHELL simply transfers control of the processor to it; when done, the builtin code returns to the SHELL.
* The code for a program is in a disk file. The compiler, assembler, linker, and the programs that you create are all command programs. To execute a program's code, the SHELL must load the code into memory, thus overlaying the SHELL. When the program is done, the SHELL must be reloaded from disk.
* An exec file is a file containing a list of commands. We're not going to discuss exec files any further in this section. For more information on them, see the chapter on the SHELL.

When a command is given to the SHELL it checks its builtin list first. If it's not found there, the SHELL then looks for a file that has the same name. This file can contain either a command program, or a sequence of commands that the SHELL is to execute.

When you want to execute a command program or an exec file, you can explicitly specify the directory containing the file. For example, the *as* assembler is in the */cc* directory. Thus, to run the assembler you could say:

```
        /cc/as
```

If you enter the name of a command program or exec file without specifying the directory containing it, the SHELL will search for a file of that name in the directories specified by the *PATH* environment variable. An "environment variable" is a variable whose character string value you define using the SHELL's *set* builtin command. For more information, see the sections in the SHELL chapter on environment variables.

5. Loading the SHELL

When the SHELL starts a program, it loads the program's code into memory from a disk file and then transfers control of the processor to the memory resident code. When the program terminates, the SHELL is restarted. The program is loaded into the memory that the SHELL occupied; thus, when the program terminates, the SHELL must be reloaded from disk into memory.

Since the SHELL is frequently reloaded from disk, it's best to have it reloaded from a disk that is usually mounted, such as a ram disk or a hard disk. If your system doesn't have a ram disk or a hard disk, it's best to have the SHELL reloaded from your "working disk" (ie, the disk on which you've placed your own files), since this disk is usually in a drive.

To redefine the file from which the SHELL is reloaded, just type the name of the file when the SHELL is waiting for you to enter a command. For example, to have the SHELL loaded from a ram disk, enter the following commands:

```
        cp /system/shell.system /ram
        /ram/shell.system
```

The first command copies *shell.system* from the */system* disk to the */ram* disk. The second command then reloads the SHELL from the ram disk. After this, when ever the SHELL needs to be reloaded, it will be reloaded from the file *shell.system* on the */ram* disk.

As implied by the above discussion, the SHELL is loaded from the file from which it was last loaded. If the disk containing this file isn't in a drive when a program terminates, a message will be displayed and you can then place the disk into a drive.

6. Device Configuration

When the SHELL that is on the */system* disk starts, it will determine the attributes of the console; then it and PRG programs will make use of these attributes. It will also assume that your system has a printer having certain attributes.

You can also explicitly define the devices that are on your system using the *config* program. For information, see the Devices section of the SHELL chapter, and the description of *config* in the Utility Programs chapter.

7. Native Code vs. Pseudo Code

Aztec C65 comes with two compilers and two assemblers: The *cc* compiler and *as* assembler, which together generate native machine code; and the *cci* compiler and *asi* assembler, which together generate pseudo code that must be interpreted.

There are advantages and disadvantages to using each compiler/assembler pair:

* Code generated by *cc* and *as* is fast but large;
* Code generated by *cci* and *asi* is small but slow.

Thus, when you are going to create an executable program, you must decide which compiler/assembler pair to use. We recommend that you first use *cc* and *as*. If it gets too large, use *cci* and *asi*. If neither of these alternatives is acceptable, with a native code version being too large and an interpreted version being too slow, you can divide the program into modules, compiling and assembling some of them into native code, the rest into interpreted code, and linking them all into a single executable program.

8. Installing Aztec C65 on a Hard Disk

To install Aztec C65 for use on a system having a hard disk, follow these steps:

1. Copy the Aztec C65 program files from the distribution disks into a directory on the hard disk. This can be done using the SHELL's *cp* command or using the ProDOS filer.

2. Copy the Aztec C65 header files from the */cc* disk into a directory on the hard disk.

3. Copy the Aztec C65 library files (c.lib, etc) from the distribution disks into a directory on the hard disk.

4. To make ProDOS automatically activate the SHELL from the hard disk during system startup, copy *shell.system* from the */system* distribution disk into the hard disk's volume directory. This must be the first *.system* file in the hard disk's volume directory.

5. Using *ved*, create a file named *profile* in the hard disk's volume directory containing commands that define frequently-used environment variables; when the SHELL starts, it will automatically execute the commands in this file. The environment variables that should be defined in the *profile* are:

PATH	Defines a sequence of directories in which the SHELL should look for command and batch files. For details, see section 2.8 of the Shell chapter.
INCLUDE	Defines directories in which the compilers and assemblers should look for files that are specified in *#include* statements. For details, see section 1.3 of the Compilers chapter.
CLIB	Defines the directory in which the linker will look for libraries that are specified using the linker's *-l* option. For details, see the discussion of this option in section 3.2.1 of the Linker chapter.

For example, suppose that your hard disk's volume directory is named */pro*, and that you want the SHELL, when it goes looking for a command or batch file, to look first in the current directory, then in a ram disk named */ram*, and finally in */pro/bin*. Suppose further that your programs are in directory */pro/bin*, that include files are in directory */pro/include*, and that libraries are in */pro/lib*. Then the *profile* file could contain the following commands:

```
set PATH=::/ram:/pro/bin
set INCLUDE=/pro/include
set CLIB=/pro/lib/
```

Note the terminating slash on the *set CLIB* command.

9. Where to go from here

In this chapter, we've just begun to describe the features of Aztec C65. You should know enough now to create some simple programs and use the SHELL, which you can do while continuing to read the rest of this manual.

In your reading, be sure to read the sections on the SHELL, compiler and linker. You should scan through the Utility Programs chapter, which describes in detail each of the builtin commands and command programs that are provided with Aztec C65.

The Technical Information chapter also discusses several topics which might be of interest to you.

THE SHELL

Chapter Contents

The SHELL

The SHELL is a program, which runs under ProDOS, that provides an efficient and convenient environment in which to develop programs.

The basic function of the SHELL is to execute commands. You enter commands by typing on the keyboard. When it finishes executing a command, the SHELL writes a prompt to the screen and waits for another command to be entered.

There are three types of commands: builtins, programs, and exec files. The operator doesn't have to specify the type of an entered command, just its name. When a command is entered, the SHELL first searches for a builtin command, and then for a program or exec file.

Builtins are commands whose code is built into the SHELL. To execute a builtin command, the SHELL simply transfers control of the processor to the command's code. When done, the command's code returns control of the processor to the main body of the SHELL.

Programs are commands whose code resides in a disk file. The name of a command is the name of the file containing its code. The SHELL executes a program by loading its code into memory, overlaying the SHELL, and then transfering control of the processor to the loaded code. When the program is done, the SHELL is automatically reloaded into memory and regains control of the processor.

Exec files are disk files containing text for a sequence of commands. The SHELL executes an exec file by executing each of the file's commands.

This chapter first discusses the file system supported by the SHELL and then describes the features of the SHELL. The *utilities* chapter describes the SHELL's builtin commands and the program commands that are provided with the Aztec C package.

1. The file system

The SHELL supports the ProDOS file system. In this section we want to describe this file system, in case you aren't familiar with it, and then briefly describe the SHELL's file-related commands.

Programs can access information contained on one or more disks, or 'volumes', as they're called in ProDOS. The information is contained in logical entities called 'files', each of which has a name. A single file is contained within one volume; that is, a file can't span several volumes.

Along with files, a file system contains directories. A directory contains a number of entries, each of which identifies a file or another directory. Files having entries in a particular directory are said to be contained in the directory, and the directories having entries in a directory are said to be subdirectories of that directory. A file is contained in exactly one directory, and a directory other than a special directory called the "root directory" is a subdirectory of exactly one directory. The root directory isn't a subdirectory of any directory.

Each volume has a special directory called the "volume directory". All directories on a volume can be reached by passing through a sequence of directories that begins with the volume's volume directory.

The volume directories of the volumes that are in disk drives, or that are otherwise known to ProDOS (for example, the ram disk), are subdirectories of the file system's root directory.

All directories, except for the root directory, have a name. The name of a file or directory must be unique within the directory that contains it, but two files or directories that are in different directories can have the same name.

An example

For example, figure 1 depicts the organization of a file system. This file system contains two volumes: one volume (whose volume directory is named *work*) is a disk in a disk drive, and the other (whose volume directory is named *ram*) is the ram disk.

The root directory for the file system contains, as subdirectories, the *work* and *ram* directories.

The *work* volume contains the files *hello.c* and *hello.o*, and the directory *subs*.

The *ram* volume contains the files *stdio.h* and *ctype.h*, and the directory *subs*. Notice that there are two directories named *subs*. We'll describe below the naming convention for directories, which will make clear how a directory is uniquely identified.

The *subs* directory that is a subdirectory of the *ram* directory contains just the file *in.c.*

The *subs* directory which is a subdirectory of the *work* directory contains two files: *in.c* and *out.c.* The *in.c* file in this directory is different from the *in.c* which is in the other *subs* directory.

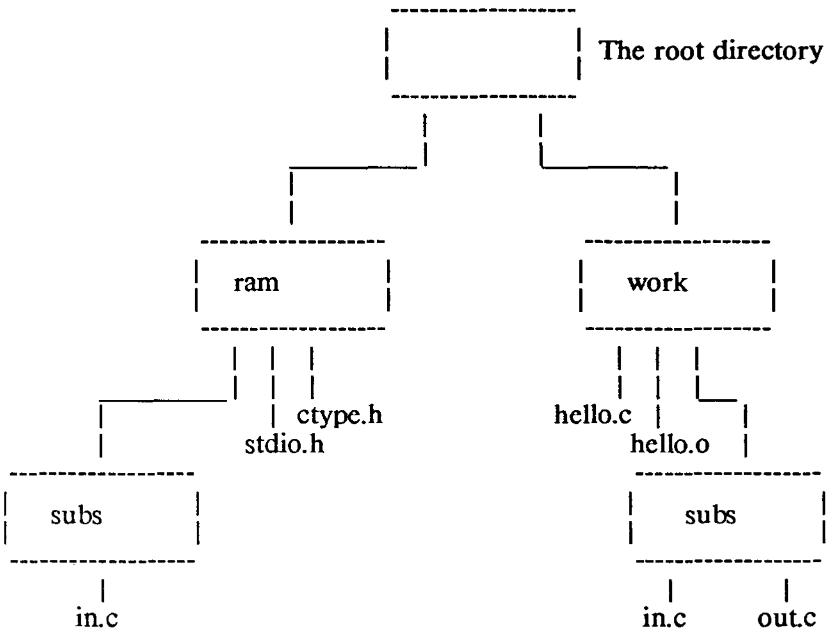

Figure 1: a sample file system

1.1 File names

There are two parts to a name that identifies a file:

* The path to the directory containing it;
* The file name itself.

For example, the file *in.c* in figure 1, which is in the *subs* directory, which is a subdirectory of the *work* directory, which is a subdirectory of the root directory, is identified by the name:

```
/work/subs/in.c
```

where */work/subs/* is the path identifier and *in.c* is the file name.

The following paragraphs describe the naming convention in detail.

File and Directory Names

A file or directory name can contain up to 15 alphabetic characters, digits, and periods. The case (upper or lower) of an alphabetic character is not significant.

By convention, the Manx programs assume that a file name contains a main part, usually called the "filename", optionally followed by a period and an extension. With this convention, related files can have the same basic filename, and different extensions. Extensions used by the Manx software are:

extension	*file contents*
.c	C source
.asm	assembler source
.o	relocatable 6502 object code
.i	relocatable pseudo-code object code
.rsm	symbol table for overlay use
.sym	symbol table for an executable file
.lst	assembler listing

By default, the file created by the linker which contains executable code has no extension.

For example, the C source code for the "hello, world" program might be put in a file named *hello.c*. The file containing the relocatable object code for this program would by default be named *hello.o*, and the file containing the executable code for the program would be named *hello.*

Path identifiers

The path component of a file name specifies the directories that must be passed through to get to the directory containing the file. It is a list of the directory names, with each pair separated by a forward slash character, /. The root directory doesn't have a name, and is represented by single slash, '/'.

For example, the paths to the directories used in figure 1 are:

/	Path to the root directory.
/ram	Path to the *ram* subdirectory of the root directory. This subdirectory is also the volume directory of the ram disk.
/ram/subs	Path to the *subs* directory that is a subdirectory of the *ram* directory;
/work	path to the *work* directory, which is a subdirectory of the root directory. This subdirectory is also the volume directory of the floppy disk that's in a disk drive.
/work/subs	Path to the *subs* directory that is a subdirectory of the *work* directory.

Each directory can be reached from the root directory by passing through a unique path of directories. This is why two directories which are subdirectories of two different directories can have the same name and still be uniquely identified: the path to each one is different.

Examples

The complete names of some of the files in figure 1 are:

```
/ram/stdio.h
/ram/subs/in.c
/work/hello.c
/work/subs/in.c
```

Frequently, the complete file name needn't be given to identify a file. The file can be located relative to a directory called the 'current directory', thus allowing the path to be omitted from the file name. This is discussed below.

1.2 The current directory

Having to specify the complete name of each file you want to access would be very cumbersome. Also, when developing programs, at any time, you are generally interested in the files on a single directory. For these reasons, the SHELL allows one directory, called the 'current directory', to be singled out.

When the SHELL is first started, the root directory on the volume containing the SHELL is the current directory; there is also a command, *cd*, which allows the operator to make another directory the current directory.

A file on or near the current directory can be specified by the operator or program without having to list the complete name of the file:

* If the name doesn't specify the path, the file is assumed to be in the current directory.
* If the name doesn't specify a path which begins at the root, the path is assumed to begin with the current directory.

For example, suppose that the current directory on the volume depicted in figure 1 is *work*. The complete name of the file *hello.c* in this directory is

```
/work/hello.c
```

Since this file is in the current directory, the operator or a program can refer to it without the path; that is, simply as

```
hello.c
```

Since the directory */work/subs* is a subdirectory of the current directory, the file *out.c* within */work/subs* can be identified with only a partial path name; that is, as

```
subs/out.c
```

1.2.1 The '.' directory

The current directory can be referred to using the character '.'. For example, the following command will copy the file *hello.c* that is in the */source* directory to the file *new.c* in the current directory:

```
cp /source/hello.c ./new.c
```

Since a file is assumed to be in the current directory unless you specify otherwise, the above command is equivalent to the following

```
cp /source/hello.c new.c
```

1.2.2 The '..' directory

The parent directory of the current directory can be specified using two periods as the path name. For example, in figure 1, with the */work/subs* directory as the current directory, the file *hello.c* could be referred to as

```
../hello.c
```

and the file *ctype.h* in the directory *ram* could be identified as:

```
../../ram/ctype.h
```

1.3 Directory-related builtin commands

The SHELL has several builtin commands for examining and manipulating directories: *pwd*, *cd*, *ls*, and *df*. We want to introduce these commands in this section; complete descriptions are presented in another section of the manual.

pwd

This command, whose name is a mnemonic for 'print working directory', displays the names of the directories that must be passed through to get to the current directory. The names are separated by a slash, '/'.

cd

This command makes another directory the current directory. If the new directory doesn't exist, the current directory remains unchanged.

The command has one argument, which specifies the directories that must be passed through to get to the desired directory. This argument has the same format as the path component of a file name.

For example, considering figure 1, with */work* being the current directory, the following *cd* commands change the current directory as indicated:

command	*new current directory*
cd /ram	/ram
cd subs	/work/subs
cd ..	/ (the root directory)

ls

ls displays the names of files and the contents of the directories whose names are passed to it.

The format is:

```
ls [-l] [name] [name] ...
```

where square brackets indicate that the enclosed field is optional.

-l causes *ls* to display information about the files or directories in addition to their names.

The *name* arguments are the names of the files and directories of interest. If no 'name' arguments are specified, the command displays information about the current directory.

For example, the following displays the names of the files and directories in the current directory:

```
ls
```

The following displays information about the files and directories in the current directory:

```
ls -l
```

The following displays the names of the files and directories contained in the */ram* directory:

```
ls /ram
```

The following displays information about the file *in.c* in the directory */john/progs*:

```
ls -l john/progs/in.c
```

For more information about the *ls* command, particularly about the information displayed when the '-l' option is used, see the description of *ls* in the utilities chapter.

1.4 Miscellaneous file-related commands

In this section we want to list the rest of the file-related commands that are built into the SHELL. For complete descriptions, see the utilities chapter.

rm	-	Remove files
cp	-	Copy files
mv	-	Move files This will either rename the files or copy them and erase the originals, depending on whether the old and new files are on the same volume.
cat	-	Display text files.
df	-	Display file information
lock/unlock	-	Lock/unlock files.

2. Using the SHELL

The previous section presented information on the SHELL's file system, which you need to know before you can use the SHELL. With that information in hand, you can continue on with this section, which shows you how to use the SHELL.

2.1 Simple Commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; the other words are arguments to be passed to the command. The name of the command is always passed to a command as an argument. For example,

```
ls
```

lists the names of the files and directories that are in the current directory. The first word on the command line, *ls*, is the name of the command. No other words are specified, so the only argument passed to the 'ls' command is the name of the command.

The *ls* command can also be passed arguments; the command

```
ls /bin
```

displays the names of the files and directories in the directory named */bin.* The first word on this command line, *ls* is the name of the command to be executed. Two words are passed to the *ls* command as arguments: *ls* and */bin/.*

The command

```
rm hello.bak temp /include/head.o
```

removes the files *hello.bak*, *temp*, and */include/head.o.* The name of this command is *rm.* Four words are passed to it as arguments: *rm*, *hello.bak*, *temp*, and *include/head.o.*

The command

```
ls -l /include
```

displays the names of the files and directories in the directory */include.* The '-l' causes the *ls* command to display other information about the files and directories in addition to their names. For this command, three words are passed to the *ls* command: *ls*, *-l*, and */include.*

The meaning of the arguments following the command name on a command line is particular to each command. Usually, either they are 'switches', indicating a particular command option, as in the *ls -l /include* command above, or they are file names. By convention, switches usually precede file names in a command line, although there are exceptions to this.

2.2 Pre-opened I/O channels

When a builtin command or command program is started by the SHELL, three I/O channels are automatically pre-opened for it by the SHELL: standard input, standard output, and standard error. By default, these channels are connected to the console, and most programs use these devices when communicating with the operator. For example, the *ls* command displays information about files and devices on the standard output channel and writes error messages to the standard error channel.

2.2.1 Standard output

The operator can request that the standard output channel be pre-opened to another file or device other than the console by including a phrase of the form '> name' on the command line . For example, the following command causes *ls* to write information about the files and directories in the current directory to the file *files.out*, instead of the console:

```
        ls > files.out
```

If the specified file doesn't exist, it is created; otherwise, it is truncated to zero length.

The standard output channel can also be redirected so that output to a file via the standard output is appended to the file. This is done by including a phrase of the form '>> file' on the command line. For example, the following command causes *ls* to append information about the files and directories in the current directory to *files.out*:

```
        ls >> files.out
```

If the specified file doesn't exist, it is created; otherwise it is opened and positioned at its end.

2.2.2 Standard input

The operator can request that the standard input device be pre-opened to a file or device other than the console by including a phrase of the form '< name' on the command line. For example, if the program *prog* reads from the standard input channel, then the command

```
        prog
```

causes *prog* to read from the console, and the command

```
        prog <names.in
```

causes it to read from the file *names.in.*

2.2.3 Standard error

A program's standard error channel can also be redirected to another file or device other than the console, by including a phrase of the form:

```
2> name
```

where *name* is the name of the device or file to which standard output is to be connected.

For example, the following causes *ls* to display the names of all files in the directory */work* having extension *.c*. The names are sent to the file *ls.out* in the current directory and any error messages are sent to the file *err.msg*:

```
ls /work/*.c >ls.out 2>.bout
```

2.2.4 Other I/O channels

Channels other than standard input, standard output, and standard error can be pre-opened for a program. The channel having file descriptor *i* is pre-opened for output to a device or file named *name* by including the phrase

```
i> name
```

on the command line. And it's pre-opened for input by including

```
i< name
```

on the command line.

For example, the following command pre-opens the channel having file descriptor 3 for output to the file *info.out*:

```
prog 3>info.out
```

2.2.5 Creating empty files

The SHELL allows you to enter a command line containing only I/O redirection components. In this case, the SHELL processes the I/O redirection clauses and then reads another command line.

Such a command line can be used for recording the time at which events occur. For example, the command

```
> mytime
```

creates an empty file named *mytime*. The *last-modified* field for this file is set to the time at which it was created.

2.3 Expansion of file name templates

When the characters '?' and/or '*' appear in a command line argument, the SHELL interprets the argument as a template to be matched to file names. Each matching name is passed to the program as a separate argument, and the template isn't passed. If the template doesn't match any file names, it is passed to the program, unaltered.

These characters can only be used within the filename component of a file name, and not the volume or path components.

2.3.1 The '?' character

The character '?' in a template matches any single character. For example, the command

```
rm ab?d
```

would remove files in the current directory whose names are four characters long, the first two being 'ab' and the last being 'd'. Thus, it would remove files with names such as

```
abcd  abxd  ab.d
```

from the current directory.

Continuing with this example, if the three files listed above were the only ones in the current directory that matched the template "ab?d", then pointers to those three names are passed to the *rm* command in place of a pointer to the template. So the *rm* command would behave as if the operator had entered

```
rm abcd abxd ab.d
```

If no files matched the template, a pointer to the template itself would have been passed to *rm*.

Notice that the template "ab?d" matches "ab.d". This emphasises the fact that extensions in file names, and their preceding period, are simply conventions and are not afforded special treatment by the SHELL, as they are in some other systems.

2.3.2 The '*' character

The character '*' matches any number of characters, even none. For example,

```
rm /work/ab*d
```

removes all files in the */work* directory whose names begin with the characters 'ab' and end with 'd'. Thus, it would match files in the */work/* directory having names such as

```
abd  abcd  ab123d  ab.exd
```

As with templates containing '?', the names of files which match a template containing '*' are passed to the program, each as a separate argument, and the template isn't passed. The template is passed only if no files match it. Thus, if the files listed above were the only ones that matched the template, then the following would have been equivalent to 'rm /work/ab*d':

```
rm /work/abd /work/abcd /work/ab123d /work/ab.exd
```

The use of '*' templates can be dangerous. For example, if you wanted to type

```
rm abc*
```

but mistyped it as

```
rm abc *
```

then *rm* will remove "abc", if it exists, and then remove all other files in the current directory.

2.4 Quoting

Characters such as *, <, and > are special, because they cause the SHELL to perform some action and are not normally passed to a program. There are occasions when you want such characters to be passed to a program without having the SHELL interpret them. This can be done by preceding the character with a backslash character, '\'. Any character can be preceded by a backslash; when the SHELL encounters '\' in a command line it removes the backslash from the line and treats the following character as a normal character, without attempting to interpret it.

For example, the command

```
echo *
```

displays the names of all files and directories in the current directory on the console. The command

```
echo \*
```

displays the character '*' on the console.

The backslash character and multi-line commands

The backslash character can also be used to enter long command lines on several physical lines. Normally, a newline character causes the SHELL to terminate the reading of a command line and to begin execution of the command. When the newline character is preceded by a backslash, the SHELL removes both characters from the command line and continues reading characters for the command line. For example,

```
echo abc\
def
```

displays 'abcdef' on the console.

When the SHELL needs additional input from the console before it can execute a command, it will prompt you with its secondary prompt. By default, this is the character '>'. The primary prompt, which is displayed when the SHELL is ready for a new command, is by default '-?'. Prompting is discussed in more detail below.

Quoted strings

A string in the command can be surrounded by single quotes. In this case, the SHELL considers the entire string within the quotes to be a single argument. The SHELL doesn't try to interpret any special characters contained in a string that is surrounded by single quotes.

For example, consider a program, *args*, which prints the arguments passed to it, each on a separate line. The command

```
args 123 234 345
```

would print

```
args
123
234
345
```

(the command name is passed to the program as an argument), while the command

```
args '123 234 345'
```

would print

```
args
123 234 345
```

The command

```
args *
```

would print the names of each of the files on the current directory, each on a separate line, while

```
args '*'
```

would print the character '*'.

A quoted string can contain newline characters. That is, if the SHELL sees a quote character and then reads a newline character before finding another quote, it will keep prompting for additional input until it finds another quote. The argument corresponding to the quoted string then consists of the string with the newline characters still imbedded in it.

For example, if you enter

```
echo 'ab
```

the SHELL will prompt you for additional input, using its secondary prompt. If you then enter

```
1
2
3'
```

the echo command will be activated with arguments

```
echo
ab\n1\n2\n3
```

(where '\n' stands for the newline character) and will print

```
ab
1
2
3
```

Double-quoted strings

A string on the command line can also be surrounded by double quotes. The only difference in the treatment of singly- and doubly-quoted strings by the SHELL is that variable substitution is done for double-quoted strings but not for single-quoted strings. This is discussed in detail in the section on environment variables.

2.5 Prompts

The SHELL prompts you when it wants you to enter information, by writing a character string, called a 'prompt' to the console. There are two types of prompts: one when the SHELL is waiting for a new command to be entered, and the other when it needs additional input before it can process a partially-entered command.

2.5.1 The primary prompt

The first type of prompt is called the 'primary' prompt. By default, it is the string '-?'. This can be changed by entering the command of the form

```
set PS1=prompt
```

where 'prompt' is the desired prompt string. For example,

```
set PS1='>>'
```

sets the primary prompt to '>>'. Note the single quotes surrounding >>. These are necessary to prevent the SHELL from trying to interpret these special characters.

```
set PS1='hi there, fred. please enter a command: '
```

sets the primary prompt to the specified, space-containing string.

2.5.2 The secondary prompt

The second type of prompt is called the 'secondary' prompt. By default, it is the string '>'. This can be changed by entering a command of the form

```
set PS2=prompt
```

2.5.3 The command logging prefix

When command logging is enabled, the SHELL logs each command to the console, and precedes it with a character string called the 'command logging prefix'. By default, this prefix is the character '+', and can be set by entering a command of the form

```
set PS3=prefix
```

2.5.4 Special substitutions

The prompts and prefix described above can contain codes that cause variable information to be included in a prompt. The codes consist of a lower case letter preceded by the character '%'. For example, to set the primary prompt to the time, followed by ' :' enter

```
set PS1='%t :'
```

The list of letters and their substituted values are:

letter	*substituted value*
d	Date
t	Time
v	Current volume
c	Current directory

2.6 The program's view of command line arguments

In this section we want to describe the passing of arguments by the SHELL to the three types of programs that the Aztec linker can create: programs of type PRG (that can be started by the SHELL but not by the Basic Interpreter); programs of type BIN (that can be started by the SHELL and by the Basic Interpreter); and system programs (that are loaded at 0x2000).

For more information on the different types of Aztec-generated programs, see the *Command Programs* section of the *Technical Information* chapter.

2.6.1 Passing Arguments to PRG Programs

The *main* function of a program is the first user-written function to be executed when the program is started. The SHELL passes two arguments to the *main* function of a program of type PRG, as follows:

```
main(argc, argv)
int argc; char *argv[];
```

argc contains the number of command line arguments passed to the program. The command itself is included in the count.

argv is an array of character pointers, each of which points to a command line argument.

For example, if the operator enters the command

```
prog abc def ghi
```

then the *argc* parameter to *main* will be set to 4, and the *argv* array is set as follows:

argv element	*points to*
0	"prog"
1	"abc"
2	"def"
3	"ghi"

As another example, for the command

```
prog "abc def ghi"
```

argc is set to 2, and the *argv* array as follows:

argv element	*points to*
0	"prog"
1	"abc def ghi"

With the command

```
prog *.c
```

and the current directory containing the files

```
a.c a.o a b.c
```

argc will be set to 5, and the *argv* array as follows:

argv element	*points to*
0	"prog"
1	"a.c"
2	"a.o"
3	"a"
4	"b.c"

2.6.2 Passing Arguments to BIN and system programs

A program that can be activated by the Basic Interpreter (that is, a program of type BIN or a system program) can also be activated by the SHELL. When the SHELL starts such a program, the first parameter of the program's *main* function (*argc*) is set to 0, and its second parameter (*argv*) is set to a null pointer.

2.7 Devices

Programs can access the following devices:

* The console, named *con:*
* A printer, named *pr:*
* A serial device, named *ser:*

For example, the following command copies the output of the *ls* command to the printer:

```
ls > pr:
```

In addition, programs can access the card in a particular slot using the name *sx:*, where *x* is the slot's number. For example, the following command copies the output of *ls* to the card in slot 2:

```
ls >s2:
```

2.7.1 Device Configuration

Using the *config* program, you can define to the SHELL the devices that are connected to your Apple. Knowledge of this configuration is then available both to the SHELL and to PRG programs that you tell the SHELL to start. You can also use *config* to define a configuration to stand-alone programs that you create using the Aztec software; that is, to ProDOS BIN and SYS programs, and to programs that run on DOS 3.3.

For details on *config* see its description in the Utility Programs chapter.

The console is one device for which you can define attributes using *config*. If the SHELL or a stand-alone program starts without your having predefined the console attributes to it using *config*, the SHELL or stand-alone program will determine the type of Apple on which it's running and set the console attributes accordingly.

Similarly, if the SHELL or a stand-alone program starts without your having predefined the printer attributes to it using *config*, the program will assume that the printer has the following attributes:

* Its card is in slot 1,
* It is initialized using the string ^I^Y^Y255N.
* Characters sent to it must have their most significant bit set;
* A carriage return character must be followed by a line feed character.

2.7.2 Console I/O on an Apple // Plus

A standard Apple // Plus does not support the full ASCII character set on keyboard input or screen output. There are hardware modifications that you can make to an Apple // Plus that provide some help, and our software assumes that you have made these modifications. One of these changes is the "single wire shift key mod", and the other is a modification that allows the console to display the full set of displayable ASCII characters. For information on these modifications, see your Apple dealer.

Even with these changes, you still can't enter the special C characters on an Apple // Plus, so our software translates certain control characters that you type into those characters. The following table lists these control characters and the characters to which they are translated. In this table, as in the rest of this manual, ^X is an abbreviation for "type X while holding the control key down". The first column identifies control codes that you type; the second identifies the characters to which control codes are translated when the SHIFT key is held down; and the last column identifies the characters to which control codes are translated when the SHIFT key is held down.

Press:	*To get (lower):*	*To get (upper):*
^P	`	@
^A	{	[
^E	|	\
^R	}	]
^N	~	^
^C	DEL	_

To enter a TAB character on an Apple // Plus, type the right arrow key that is on the far right of the Apple keyboard.

2.7.3 Other special control keys

Regardless of the type of console you are using, several control characters that you type have special meaning:

^C	Causes the program to halt and return control to the command processor program (ie, to the SHELL or the Basic Interpreter); A check for ^C is made both when a program is reading from the keyboard and when it is writing to the screen.
^S	Causes screen output to be suspended until you type another ^S.
^D	Causes EOF to be sent to a program that is reading the keyboard.
^H	Moves the cursor one character to the left on the screen. When the SHELL has requested input, it also erases that character from the screen and from the SHELL's input buffer. The SHELL reads characters into this buffer when its waiting for a command and then executes the command when you type the RETURN key.
DEL	Same as ^H.
^X	Causes the SHELL to clear its input buffer and move

the cursor to the next line on the screen. Thus, ^X essentially deletes the command line that you are currently typing.

RETURN When you type RETURN, the keyboard input routine translates it to a Newline character.

2.8 Exec files

An "exec file" is a file containing a sequence of commands. The operator causes the SHELL to execute the commands in an exec file by simply typing its name.

For example, if the file named *dir* in the current directory contains the commands

```
pwd
ls -l
```

then when the operator types

```
dir
```

the SHELL will execute the commands *pwd* and *ls -l*.

An exec file can contain any command that can be entered from the console. In particular, an exec file can execute another exec file; that is, exec files can be chained. However, when one exec file calls another, control never returns to the calling exec file; that is, exec files cannot be nested.

2.8.1 Exec file arguments

The command line that activates an exec file looks just like a command line that activates a builtin or program command. Exec files can be passed arguments in the same way that builtin and program commands are passed arguments:

* a space-delimited string is normally passed to the exec file as a single argument;
* A quoted string is passed as a single argument;
* Filename-matching templates, containing '?' and '*', are replaced, when a match is made, by the matching file names;
* '\' causes the next character to be passed to the exec file without interpretation, and the '\' isn't passed. '\\' is replaced by a single backslash character.

The method by which an exec file accesses command line arguments is necessarily different from that used by builtin and program commands, since the exec file is not a program. The exec file can be passed any number of arguments, and it refers to them as $1, $2, ..., where $1 represents the first argument, $2 the second, and so on. $0 refers to the name of the exec file.

Before executing a command in an exec file, the SHELL replaces the $x variables with the corresponding command line arguments. $x variables which don't have a corresponding argument are replaced by the null string.

For example, the following exec file displays the value of the first, fourth, and ninth arguments, and the name of the command itself, each on a separate line:

```
echo the first argument is $1
echo the fourth argument is $4
echo the ninth argument is $9
echo and me, I'm $0
```

If the exec file is named *names* then

```
names a b c d e f g h i j
```

would print

```
the first argument is a
the fourth argument is d
the ninth argument is i
and me, I'm names
```

and the command

```
names *
```

would display the names of the first, fourth, and ninth files in the current directory, and the name of the command.

The command

```
names "this is one argument"
```

would print

```
the first argument is this is one argument
```

The $# variable

Several other variables are set when an exec file is activated. $# is set to the number of arguments that were passed to the exec file. For example, an exec file named *hello* might contain

```
echo My name is $0
echo I was run with $# arguments
```

Typing

```
hello one two three
```

would print

```
My name is hello
I was run with 3 arguments
```

The $* and $@ variables

$* and $@ are two other variables that are set when an exec file is activated. Both of these are set to a character string consisting of all the exec file's arguments, less $0. For example, consider an exec file *allargs*, which contains

```
args $*
```

where *args* is a command program that prints its arguments, each on a separate line. Typing

```
allargs one two three
```

would give

```
args
one
two
three
```

Exec file variables and quoted strings

When an exec file variable is contained within a character string surrounded by single quotes, the SHELL does not replace the variables with their values. Thus, given the exec file *info*, which contains

```
echo 'number of args = $0'
echo 'args = $0 $1 $2'
echo 'all args = $* and $@'
```

then typing

```
info one two three
```

gives

```
number of args = $0
args = $0 $1 $2
all args = $* and $@
```

As mentioned in section 2, the SHELL does substitute variables that are contained within character strings that are surrounded by double quotes. Thus, the exec file

```
args "$*"
```

will pass the exec file arguments to echo as a single argument and is equivalent to

```
args "$1 $2 $3 ..."
```

$* and $@ are the same, except when surrounded by double quotes.

The exec file

```
args "$@"
```

is equivalent to

```
args "$1" "$2" ...
```

2.8.2 Exec file options

There are three options related to exec files: logging of exec file commands to the screen, continuation of an exec file following execution of a command which terminates with a non-zero exit code, and execution of commands.

Each option has an identifying character. An option's value is set by issuing a *set* command, giving the option's character preceded by a minus or plus sign. Minus enables an option and plus disables it.

The options, their identifying characters, and their default values are listed below:

character	*option*	*default*
x	log commands	disabled
e	abort on non-zero	enabled
n	don't execute cmds	disabled

Several options can be enabled or disabled in a single *set* command, and an exec file can contain several option-setting commands.

The same *set* command is used to set exec file options and to set environment variable values. *set* commands which set environment variables can also be contained in an exec file. However, a single *set* command cannot set both environment variables and exec file options.

When the SHELL logs exec file commands to the console, it precedes each command line with the character '+'. This prefix can be changed by entering a command of the form

```
set PS3='string'
```

where 'string' is the desired prefix.

The following are valid *set* commands for manipulating exec file options:

set -x	enable logging
set +x	disable logging
set -x -n	enable logging and non-execution of cmds
set -x +e	enable logging, disable return code chk

Exec file options are inherited by a called exec file. That is, if you type

```
set -x
docmds
```

where *docmds* is an exec file, the 'x' option is enabled in *docmds*.

An exec file can change the setting of the exec file options, but these changes don't affect the settings of the options in the caller. Thus, if *docmds* includes the command

```
set +x
```

then the 'x' option will be disabled during the execution of *docmds*, but when control returns to the operator, the 'x' option is reenabled.

2.8.3 Comments

In an exec file, any line beginning with the character '#' is considered to be a comment, and is not executed. Argument substitution is performed on it, though, allowing exec files like:

```
set -x
# the first arg is $1
# the second is $2
```

2.8.4 Loops

Exec files can contain 'loops'; that is, sequences of commands that are executed repeatedly, each time with an environment variable assigned a different value.

A loop has the format

```
loop
cmdlist
eloop
```

where *cmdlist* is the sequence of commands. The SHELL will repeatedly execute the *cmdlist* commands; after each pass through the commands it will shift down the exec file's arguments, so that argument 2 becomes argument 1, argument 3 becomes argument 2, and so on. When the argument list becomes empty, the SHELL will exit the loop and execute the command that follows the *eloop.*

For example, the following exec file compiles the C source files whose names are passed to it (without the ".c" extension):

```
loop
echo compiling $1
cc $1
eloop
echo "*** all done***"
```

2.8.5 The *shift* command

The command

```
shift
```

causes the exec file variable $1 to be assigned the value of $2, $2 to be assigned the value of $3, and so on. The original value assigned to $1 is lost. When all arguments to the exec file have been shifted out, $1 is assigned the null string.

For example, the following exec file, *del,* is passed a directory as its first argument and the names of files within the directory that are to be removed:

```
set j = $1
shift
loop
rm $j/$1
eloop
```

In this example, 'j' is an environment variable. Environment variables are described in the section on environment variables, so you may want to reread this section after reading that section.

The first two statements in the exec file save the name of the directory and then shift the directory name out of the exec file variables.

The loop then repeatedly calls *rm* to remove one of the specified files from the directory.

Entering

```
del /work file1.bak file2.bak
```

will remove the files *file1.bak* and *file2.bak* from the */work* directory.

2.9 Environment variables

An environment variable is a variable having a name and having a character string as its value. Environment variables have two functions:

* They can be used to pass information to a program;
* They can be used to represent character strings within command lines.

Information can also be passed to programs as command line arguments, as described in a previous section.

2.9.1 Defining environment variables

Environment variables can be created by the operator, using the *set* command, and retain their value until changed by another *set* command. In particular, environment variables retain their existence and values even when programs are executed.

Environment variables are case-sensitive, so the variable named *VAR* is different from one named *Var.*

The format of the *set* command which sets the value of an environment variable is:

```
set VAR=string
```

where *VAR* is the name of the variable, and *string* is the character string to be assigned to it. *string* can be null, in which case the specified variable is deleted. The variable will be created, if it didn't previously exist.

For example, to set the environment named *PATH* to the string ":/cc/bin:/progs" the following command would be used:

```
set PATH=:/cc/bin:/progs
```

To delete the PATH variable, the following command would be used:

```
set PATH=
```

Environment variables can be assigned quoted strings:

```
set NAMES='Penelope Matilda Esmarelda'
```

The *set* command, when issued without any arguments, will display the names and values of the environment variables.

The *set* command can also be used within exec files to set exec file options. This use of the *set* command is discussed in the exec file section of this chapter.

2.9.2 Passing environment variables to programs

A program can fetch the value of an environment variable using the *getenv* function, passing to it the name of the variable. Programs cannot change the value of an environment variable.

2.9.3 Use of environment variables in command lines

When the SHELL finds an environment variable name in a command line, preceded by the character '$', it replaces the name and the '$' with the value of the variable.

For example, if the environment variable *color* has the value *violet*, then entering

```
echo $color
```

is equivalent to entering

```
echo violet
```

and results in the displaying of

```
violet
```

on the screen.

As another example, given the environment variable *b*, having value '*/fred/bin/*', the following command will move the file *pgm* from the current directory to the directory */fred/bin*:

```
mv pgm $b
```

The use of environment variables isn't restricted to command line arguments. For example, given the environment variable *cmd*, having value '*ls -l /usr/math/lib/*', the following command will list the contents of the directory */usr/math/lib*:

```
$cmd
```

Environment variables names that are used in command lines can be surrounded by { and } to prevent ambiguity in cases where the variable is immediately followed by a character string. For example, if the following environment variables are defined

```
user=fred
userdy=john
```

then

```
echo ${user}
```

is equivalent to

```
echo $user
```

and displays

```
fred
```

Entering

```
        echo $userdy
```

will display

```
        john
```

since the SHELL interprets the entire string following $ to be the name of the variable. And entering

```
        ${user}dy
```

will display

```
        freddy
```

since the SHELL assumes that the environment variable name is contained in the braces.

2.9.4 Standard environment variables

A few environment variables are created and assigned initial values by the SHELL when it is first activated. These are described in the section on starting the SHELL.

2.10 Searching for commands

When the operator enters a command, the SHELL first checks to see whether it is a builtin command. If so, the SHELL executes it. Otherwise, the command must be the name of a file to be executed, so the SHELL attempts to find the file.

2.10.1 Searching for command files

The SHELL will look for a command file in the directories that are specified in the *PATH* environment variable. *PATH* consists of the directories to be searched, separated by colons. Thus, the following command will cause the SHELL to search for commands first in directory *dir1*, then in directory *dir2*, ..., and finally in directory *dirn*, issue the command

```
set PATH=dir1:dir2: ... :dirn
```

If an entry doesn't specify a complete path (that is, doesn't begin with the root directory), the path to the directory begins at the current directory. And if the entry is null, the entry specifies the current directory. The "current directory" is the directory that is current when the SHELL attempts to find a command, and not when the *set PATH* command is entered.

For example, the following command will cause the SHELL to search the current directory, then the directory */ram/bin*, and finally the directory *progs*, which is a subdirectory of the current directory.

```
set PATH=:/ram/bin:progs
```

The next command causes the SHELL to search the directory */system/bin*, then the */cmds* subdirectory of the current directory, and finally the current directory:

```
set PATH=/system/bin:cmds::
```

To display the value of all the environment variables, including *PATH*, enter the *set* command by itself; eg,

```
set
```

By default, *PATH* is set so that the SHELL will search for commands first in the current directory and then if your system has a ram disk, in the volume directory of the ram disk.

2.10.2 Program or exec file?

When the SHELL finds a file that matches the name that the operator entered, it has to decide whether it contains a program or is an exec file. It bases its decision on the file's type: if it is *TXT*, then its assumed to be an exec file; if its type is *PRG* it's assumed to contain a program.

2.11 Starting the SHELL

The SHELL can be started in several ways:

* By ProDOS, when ProDOS is itself started;
* By the Basic Interpreter, at your command;
* By a loader that is activated when a SHELL-activated program terminates;
* By the SHELL itself, at your command.

The following paragraphs discuss each of these ways of starting the SHELL.

2.11.1 ProDOS activation of the SHELL

When you turn on the Apple or type the appropriate reset keys, a bootstrap loader is loaded from the first two sectors on the disk that's in the Apple's boot drive. This loader then loads ProDOS into the high part of memory from the first file in volume directory of the disk in the boot drive whose name is *ProDOS* and whose type is SYS. ProDOS then loads and tranfers control of the processor to a command processor program; that is, a program to which you will enter commands. ProDOS loads this program from the first file in the volume directory of the disk in the boot drive whose name ends in *.system* and whose type is SYS.

The distribution disk that's labeled */system* is "bootable", as are copies that you make of it: the disk contains a bootstrap loader, ProDOS in the file named *ProDOS*, and the SHELL in the file named *shell.system*. Thus, when you turn on the Apple or hit the appropriate reset keys with this disk in the Apple's boot drive, ProDOS and the SHELL are automatically loaded and started.

2.11.2 Starting the SHELL from the Basic Interpreter

With the Basic Interpreter running, you can start the SHELL by entering a command consisting of the name of the file that contains the SHELL, preceded by a dash character:

```
-shell.system
```

This loads the SHELL into memory below ProDOS (which is always in memory), overlaying the Basic Interpreter and any basic program that was in memory.

2.11.3 Restarting the SHELL when a Program Stops

There are two parts to the SHELL: a transient section and a memory-resident "environment" section. When the SHELL starts another program, the SHELL's transient section is overlayed by the program, but its environment section usually isn't. When a SHELL-started program terminates, the transient section of the SHELL needs to be reloaded, but the memory-resident section need not be, unless it has been destroyed.

The memory-resident "environment" section of the SHELL contains information, such as environment variables, that the SHELL wants to preserve during the execution of another program. It also contains a small loader routine.

When a SHELL-activated program terminates, control is passed to the loader routine in the SHELL's environment section. This routine loads the SHELL's transient section into memory, thus overlaying the program that was active, and then transfers control of the processor to the SHELL.

The file from which the SHELL is loaded is named *shell.system*; the path to the directory containing this file is defined in a field within the SHELL's environment section. We'll describe how this field is set below.

2.11.3.1 Destruction of the SHELL's environment section

The SHELL's environment section is located in the area of memory just below 0xBF00. Programs of type PRG or BIN that you create using the Aztec C65 linker and libraries will not modify the SHELL's environment section.

When the SHELL starts a program, it sets the Applesoft and Integer Basic HIMEM fields to the base of the SHELL's environment section. Thus, even if a program started by the SHELL hasn't been created using Aztec software, the program won't destroy the SHELL's environment section, if the program respects the HIMEM fields by not modifying memory above this address. For example, the *filer* is an example of a program that wan't created using Aztec software, that can be started by the SHELL, and that won't destroy the SHELL's environment section.

System programs, which are programs whose starting address is 0x2000, are assumed to use all memory below 0xBF00, which is the first location that ProDOS uses. Accordingly, when a system program is activated, including one created using the Aztec software, it usually destroys the SHELL's environment section.

When the SHELL is restarted, it can tell if the area of memory in which it stores its environment section contains in fact a valid environment section. The SHELL will initialize this section of memory only if the area doesn't contain a valid environment section.

2.11.4 Starting the SHELL from the SHELL

The SHELL is a program, and so you can start the SHELL just as you would any other program while the SHELL is active; that is, by entering the name of the file that contains it.

The SHELL is a system program; however, when it starts itself the environment section of the original SHELL is not destroyed.

2.11.5 The SHELL's Startup Procedure

When the SHELL starts, it first checks to see whether its environment section is in memory, by testing the area of memory where it should be for known values. If the environment section is not in memory, the SHELL creates a new one.

The SHELL next sets the field in the directory that defines the path to the directory from which the SHELL will be reloaded to the path to the directory from which the SHELL was just loaded.

If the SHELL created a new environment section, it next initializes some environment variables, defines the device configuration if one was not predefined, executes the commands in the *profile*, and goes into a loop, reading and processing commands.

For information on device configuration, see the Devices section of this chapter and the description of the *config* program in the Utility Programs chapter.

2.11.6 Defining the SHELL's Startup Directory

When you develop programs using Aztec C65, you will usually swap disks in and out of disk drives as you execute the different Aztec programs. For example, you may have one disk for initially starting the SHELL, that contains the bootstrap loaded, ProDOS, the SHELL, and perhaps the *filer*; another disk that contains the native code compiler and assembler, another that contains the interpretive compiler and assembler, another that contains the linker and libraries, and another containing your own files.

From the above discussion, you know that when a SHELL-activated program terminates, the SHELL will be reloaded from the file from which it was last loaded, and that the disk containing this file must be in a drive when the SHELL-activated program terminates.

Thus, it's best to have the file from which the SHELL will be reloaded following program termination on a disk that is usually in a disk drive. The Aztec disks don't meet this criteria, since you are frequently swapping them in and out of drives; a better place is a disk that contains your own files; and the best place is the ram disk, if your system has one.

You usually won't want the disk from which the SHELL is reloaded following program termination to contain ProDOS, since it will take up space that could be used by your own files. Hence, the disk from which the SHELL is initially loaded when the Apple is turned on is different from the disk from which the SHELL is reloaded following program termination.

But following program termination, the SHELL is reloaded from the file from which it was previously loaded. So once the SHELL is initially loaded following powering on of the Apple, you must have the

SHELL start itself, in order to redefine to the SHELL the identity of the file from which the SHELL will be reloaded following program termination. For example, you could put a boot disk in the boot drive and turn on the Apple to start ProDOS and the SHELL. Then, with your own disk, named */work* in another drive, you could copy the SHELL to this disk and define this disk as containing the file from which the SHELL should be reloaded by entering:

```
cp shell.system /work/shell.system
/work/shell.system
```

2.11.7 Executing the *profile*

When the SHELL is initially started (ie, following power-up on the Apple), it will automatically search the directory from which it was loaded for a file named *profile.* If such a file is found, the SHELL will assume that it is an exec file and will execute its commands.

For example, the *profile* could create environment variables, copy *shell.system* to the ram disk, or change the default values assigned to the SHELL-created variables.

2.11.8 Initial environment variables

A few environment variables are created and assigned initial values by the SHELL when it is first activated. These are:

PATH	-	Defines the directories to be searched for a command line. If your Apple has a ram disk, PATH is initially set to *.:/ram*, which causes the SHELL to look for a command first in the current directory and then in the ram disk. If your Apple doesn't have a ram disk, PATH is initially set to .:, which cause the SHELL to look for commands just in the current directory.
PS1	-	Primary prompt. Initially set to '-? '.
PS2	-	Secondary prompt. Initially set to '>'.
PS3	-	Cmd logging string. Initially set to '+'.
HOME	-	The volume directory of the disk from which the SHELL was loaded.

You can change the values of these variables just as you would any other environment variable.

2.12 Error codes

When the SHELL detects an error, it says so with a message that usually contains a numerical code that defines the error. These are ProDOS error codes, and are defined in the following table:

hex code	*Meaning*
00	No error
01	Invalid number for system call
04	Invalid param count for system call
25	Interrupt vector table full
27	I/O Error
28	No device connected/detected
2b	Disk write protected
2e	Disk switched
40	Invalid characters in pathname
42	File control block table full
43	Invalid reference number
44	Directory not found
45	Volume not found
46	File not found
47	Duplicate file name
48	Volume Full
49	Volume directory full
4a	Incompatible file format
4b	Unsupported storage type
4c	End of file encountered
4d	Position out of range
4e	File Access error; eg, file locked
50	File is open
51	Directory structure damaged
52	Not a ProDOS disk
53	Invalid system call parameter
55	Volume control block table full
56	Bad buffer address
57	Duplicate volume
5a	Invalid address in bit map

THE COMPILERS

Chapter Contents

The Compilers

This chapter describes *cc* and *cci*, the Aztec C compilers for Apple // ProDOS. It is not intented to be a complete guide to the C language; for that, you must consult other texts. One such text is *The C Programming Language*, by Kernighan and Ritchie. The compilers were implemented according to the language description in the Kernighan and Ritchie book.

cc translates C source code into native 6502 assembly language source code. *cci* translates C source code into assembly language source for a "pseudo machine"; in an executable program, *cci*-compiled code must be interpreted by a special Aztec C routine.

This description of the compilers is divided into four subsections: which describe how to use the compilers, compiler options, information related to the writing of programs, and error processing.

To the operator and programmer, the two compilers are very similar. In the discussion that follows, we will use the name *cc* when describing features that are common to both compilers. Where differences exist, we will say so.

1. Compiler Operating Instructions

cc is invoked by a command of the form:

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

where [-options] specify 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 the input file, translates them to assembly language source, and writes the result to another file.

When the compiler is done, it by default activates the *as* assembler (*cci* by default starts the *asi* assembler). The assembler translates the assembly language source to relocatable object code, writes the result to another file, and deletes the assembly language source file. The *-A* option tells the compiler not to start the assembler.

1.1 The C source file

The extension on the source file name is optional. If not specified, it's assumed to be *.c*. For example, with the following command, the compiler will assume the file name is *text.c*:

```
        cc text
```

The compiler will append *.c* to the source file name only if it doesn't find a period in the file name. So if the name of the source file really doesn't have an extension, you must compile it like this:

```
        cc filename.
```

The period in the name prevents the compiler from appending *.c* to the name.

1.2 The output files

1.2.1 Creating an object code file

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

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

```
        cc prog.c
```

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

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

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

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

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

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

```
/RAM/TEMP/
```

then the temporary file is placed in the TEMP directory on the RAM volume.

For a description on the setting of environment variables, see the SHELL chapter.

If you are interested in the assembly language source, but still want the compiler to start the assembler, specify the option -T when you start the compiler. This will cause the compiler to send the assembly language source to a file whose name is derived from that of the file containing the C source by changing its extension to *.asm*. The C source statements will be included as comments in the assembly language source. For example, the command

```
cc -T prog.c
```

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

1.2.2 Creating just an assembly language file

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

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

If you don't use the *-O* option but do use the *-A* option, the compiler will send the assembly language source to a file whose name is derived from that of the C source file by changing the extension to *.asm* and place this file 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's in *prog.c*, sending the assembly language source to *prog.asm*.

As another example, the command

```
cc -A -O temp.a65 prog.c
```

compiles, without assembling, the C source that's in *prog.c*, sending the assembly language source to the file *temp.a65*.

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

1.3 Searching for *#include* files

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

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

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

1.3.1 The -I option.

A -I option defines a single directory to be searched. The area descriptor follows the -I, with no intervening blanks. For example, the following *-I* option tells the compiler to search the *include* directory on the *ram* volume:

```
-I/ram/include
```

1.3.2 The INCLUDE environment variable.

The INCLUDE environment variable also defines a directory to be searched for #include files. For example, the following command sets *INCLUDE* so that the compiler will search for include files in the directory */ram/include*:

```
set INCLUDE=/ram/include
```

See the SHELL chapter for details on the setting of environment variables.

1.3.3 The search order for include files

Directories are searched in the following order:

1. If the #include statement delimited the file name with the double quote character, ", the current directory on the default drive is searched. If delimited by angle brackets, < and >, this area isn't automatically searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directory defined in the INCLUDE environment variable is searched.

2. Compiler Options

There are two types of options in Aztec C compilers: machine independent and machine dependent. The machine-independent options are provided on all Aztec C compilers. They are identified by a leading minus sign.

The Aztec C compiler for each target system has its own, machine-dependent, options. Such options are identified by a leading plus sign.

The following paragraphs first summarize the compiler options and then describe them in detail.

2.1 Summary of options

2.1.1 Machine-independent Options

-A	Don't start the assembler when compilation is done.
-Dsymbol[=value]	Define a symbol to the preprocessor.
-Idir	Search the directory named *dir* for #include files.
-O file	Send output to *file*.
-S	Don't print warning messages.
-T	Include C source statements in the assembly code output as comments. Each source statement appears before the assembly code it generates.
-B	Don't pause after every fifth error to ask if the compiler should continue. See the Errors subsection for details.
-Enum	Use an expression table having *num* entries.
-Lnum	Use a local symbol table having *num* entries.
-Ynum	Use a case table having *num* entries.
-Znum	Use a literal table having *num* bytes.

2.1.2 Special Options for the ProDOS Compilers

+C	Generate 65C02 code (*cc* only).
+B	Don't generate the statement "public .begin".
+L	Turn automatic variables into statics (*cc* only).

2.2 Detailed description of the options

2.2.1 Machine-independent options

The -D Option (Define a macro)

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

```
        cc -Dmacro[=text] prog.c
```

For example,

```
        cc -DMAXLEN=1000 prog.c
```

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

```
        #define   MAXLEN   1000
```

Since the *-D* option causes a symbol to be defined for the preprocessor, this 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 -dDEBUG program.c
```

When no substitution text is specified, the symbol is defined to have the numerical value 1.

The -I Option (Include another source file)

The *-I* option causes the compiler to search in a specified directory for files included in the source code. The name of the directory immediately follows the -I, with no intervening spaces. For more details, see the Compiler Operating Instructions, above.

The -S Option (Be Silent)

The compiler considers some errors to be genuine errors and others to be possible errors. For the first type of error, the compiler always generates an error message. For the second, it generates a warning message. The *-S* option causes the compiler to not print warning messages.

2.2.1.1 The Local Symbol Table and the -L Option

When the compiler begins processing a compound statement, such as the body of a function or the body of a *for* loop, it makes entries about the statement's local symbols in the local symbol table, and removes the entries when it finishes processing the statement. If the table overflows, the compiler will display a message and stop.

By default, the local symbol table contains 40 entries. Each entry is 26 bytes long; thus by default the table contains 640 bytes.

You can explicitly define the number of entries in the local symbol table using the *-L* option. The number of entries immediately follows the *-L*, with no intervening spaces. For example, the following compilation will use a table of 75 entries, or almost 2000 bytes:

```
cc -L75 program.c
```

2.2.1.2 The Expression Table and the -E Option

The compiler uses the expression table to process an expression. When the compiler completes its processing of an expression, it frees all space in this table, thus making the entire table available for the processing of the next expression. If the expression table overflows, the compiler will generate error number 36, "no more expression space", and halt.

By default, the expression table contains 80 entries. Each entry is 14 bytes long; thus by default the table contains 1120 bytes.

You can explicitly define the number of entries in the expression table using the *-E* option. The number of entries immediately follows the *-E*, with no intervening spaces. For example, the following compilation will use a table of 20 entries:

```
cc -E20 program.c
```

2.2.1.3 The Case Table and the -Y Option

The compiler uses the case table to process a switch statement, making entries in the table for the statement's cases. When it completes its processing of a switch statement, it frees up the entries for that switch. If this table overflows, the compiler will display error 76 and halt.

For example, the following will use a maximum of four entries in the case table:

```
        switch (a) {
        case 0:                 /* one */
           a += 1;
           break;
        case 1:                 /* two */
           switch (x) {
           case 'a':            /* three */
               func1 (a);
               break;
           case 'b':            /* four */
               func2 (b);
               break;
           }                /* release the last two */
           a = 5;
        case 3:               /* total ends at three */
           func2 (a);
           break;
        }
```

By default, the table contains 100 entries. Each entry is four bytes long; thus by default, the table occupies 400 bytes.

You can explicitly define the number of entries in the case table using the compiler's *-Y* option. The number of entries immediately follows the *-Y*, with no intervening spaces. For example, the following compilation uses a case table having 50 entries:

```
cc -Y50 file
```

2.2.1.4 The String Table and the -Z Option

When the compiler encounters a "literal" (that is, a character string), it places the string in the literal table. If this table overflows, the compiler will display error 2, "string space exhausted", and halt.

By default, the literal table contains 2000 bytes.

You can explicitly define the number of bytes in this table using the compiler's *-Z* option. The number of bytes immediately follows the *-Z*, with no intervening spaces. For example, the following command will reserve 3000 bytes for the string table:

```
cc -Z3000 file
```

2.2.1.5 The Macro/Global Symbol Table

The compiler stores information about a program's macros and global symbols in the Macro/Global Symbol Table. This table is located in memory above all the other tables used by the compiler. Its size is set after all the other tables have been set, and hence can't be set by you. If this table overflows, the compiler will display the message "Out of Memory!" and halt. You must recompile, using smaller sizes for the other tables.

2.2.2 ProDOS Options

2.2.2.1 The +C Option (Generate 65C02 code - *cc* only)

The +C option causes *cc* to generate assembler source for a 65C02 processor. If this option isn't used, *cc* will generate code for a 6502 processor.

2.2.2.2 The +B Option (Don't generate reference to .begin)

Normally when compiling a module, the compilers generate a reference to the entry point named *.begin.* Then when the module is linked into a program, the reference causes the linker to include in the program the library module that contains *.begin.*

The *+B* option prevents the compilers from generating this reference.

For example, if you want to provide your own entry point for a program, and its name isn't *.begin,* you should compile the program's modules with the *+B* option. If you don't, then the program will be bigger than necessary, since it will contain your entry point module and the standard entry point module. In addition, the linker by default sets at the program's base address a jump instruction to the program's entry point; if it finds entry points in several modules, it will set the jump to the last one encountered.

2.2.2.3 The +L Option (Turn Autos into Statics - *cc* only)

The *+L* option causes the compiler to change the class of variables whose class is automatic to static. This can cause a significant increase in execution speed, since it is faster to address static variables, which are directly addressable, than automatic variables, which are on the stack and must be indirectly addressed.

Automatic variables that are declared using the *auto* keyword, (for example *auto int i*), aren't affected by the *+L* option: they will remain automatic.

Also, if a register is available for an automatic variable that is declared using the *register* keyword (for example, *register int i*), the variable will be placed in a register and will not be turned into a static. If a register is not available, however, such a variable will be turned into a static variable.

Like any other static data, an auto-turned-static is initialized to zero before the program begins.

A function that recursively calls itself may not work correctly when it is compiled with the *+L* option. For example, the following program will print 1 when compiled without the *+L* option, and 100 when compiled with the *+L* option:

```
main()
{
        printf("%d", qtest());
}

qtest()
{
   int i;

   if (++i < 100)
      qtest(i);
   return (i);
}
```

3. Writing programs

The previous sections of this description of the compiler discussed operational features of the compiler; that is, presented information that an operator would use to compile a C program. In this section, we want to present information of interest to those who are actually writing programs.

3.1 Supported Language Features

Aztec C supports the entire C language as defined in *The C Programming Language* by Kernighan and Ritchie. This now includes the bit field data type.

The following paragraphs describe features of the standard C language that are supported by Aztec C but that aren't described in the K & R text.

3.2 Structure assignment

Aztec C supports structure assignment. With this feature, a program can cause one structure to be copied into another using the assignment operator.

For example, if *s1* and *s2* are structures of the same type, you can say:

```
        s1 = s2;
```

thus causing the contents of structure s1 to be copied into structure s2.

Unlike other operators, the assignment operator doesn't have a value when it's used to copy a structure. Thus, you can't say things like "a = b = c", or "(a=b).fld" when a, b, and c are structures.

3.3 Line continuation

If the compiler finds a source line whose last character is a backslash, \, it will consider 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 array[]="ab\
        cd\
        ef";
```

3.4 The *void* data type

Functions that don't return a value can be declared to return a *void*. This provides a safety check on the use of such functions: if a *void* function attempts to return a value, or if a function tries to use the value returned by a *void* function, the compiler will generate an error message.

Variables can be declared to point to a *void*, and functions can be declared as returning a pointer to a *void*.

Unlike other pointers, a pointer to a *void* can be assigned to a pointer to any type of object, and vice versa. For other types of pointers, the compiler will generate a warning message if an attempt is made to assign one pointer to another, when the types of objects pointed at by the two pointers differ.

That is, the compiler will generate a warning message for the assignment statement in the following program:

```
main()
{
    char *cp;
    int *ip;
    ip = cp;
}
```

The compiler won't complain about the following program:

```
main()
{
    char *cp;
    void *getbuf();
    cp = getbuf();
}
```

3.5 Special symbols

Aztec C supports the following symbols:

__FILE__	Name of the file being compiled. This is a character string.
__LINE__	Number of the line currently being compiled. This is an integer.
__FUNC__	Name of the function currently being compiled. This is a character string.

In case you can't tell, these symbols begin and end with two underscore characters.

For example,

```
printf("file= %s\n", __FILE__);
printf("line= %d\n", __LINE__);
printf("func=%s\n", __FUNC__);
```

3.6 String merging

The compiler will merge adjacent character strings. For example,

```
printf("file=" __FILE__ " line= %d func= " __FUNC__,
        __LINE__);
```

3.7 Long names

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

3.8 Reserved words

const, *signed*, and *volatile* are reserved keywords, and must not be used as symbol names in your programs.

3.9 Global variables

Aztec C supports the rule of the standard C language regarding global variables that are to be accessed by several modules. This rule requires that in the modules that want to access such a variable, exactly one module declare it without the *extern* keyword and all others declare it with the *extern* keyword.

Previous versions of Aztec C did not strictly enforce this rule. In these versions, the following modified version of the rule was enforced:

* multiple modules could declare the same variable, with the *extern* keyword being optional;
* when several modules declared a variable without using the *extern* keyword, the amount of space reserved for the variable was set to the largest size specified by the various declarations;
* when one module declared a variable using the *extern* keyword, at least one other module must have declared the variable without using the *extern* keyword;
* at most one module could specify an initial value for a global variable;
* when a module specified an initial value for a global variable, the amount of storage reserved for the variable was set to the amount specified in the declaration that specified an initial value, regardless of the amounts specified in the other declarations.

In order both to enforce the standard C rule regarding global variables and to provide compatibility with previous versions of Aztec C, the current Aztec linker will generate code consistent with the previous versions, but will by default generate a "multiply defined symbol" message when multiple modules are found that declare a global variable without the *extern* keyword. The *-M* linker option can be used to cause the linker to treat global variables just as they were in previous versions of Aztec C; in this case, the "multiply defined symbol" message won't occur when several modules declare the same variable without the *extern* keyword, as long as no more than one specifies an initial value for the variable. If multiple modules declare an initial value for the same variable this message will be issued, regardless of the use of the *-M* option.

Both previous and the current versions of Aztec C prevent a global symbol from being both a variable name and a function name. When such a situation arises, the linker will issue the "multiply defined symbol" message, regardless of the use of the *-M* option.

If you have programs whose modules follow the modified version of the rule regarding global variables, and you either want to link the modules using the Aztec linker without having to specify the -M linker option and without having the "multiply defined symbols" message appear, the compiler's *-U* option can be useful. When a module is compiled with this option, all the declarations of global variables that don't specify an initial value are implicitly turned into *extern* declarations. Thus, you can place the declarations of a program's global but uninitialized variables into one file, place *#include* statements for that file in the modules that need those variables, and compile one of the modules without the *-U* option, and the others with it.

3.10 Data formats

3.10.1 char

Variables of type *char* are one byte long, and can be signed or unsigned. By default, a *char* variable is unsigned.

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

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

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

3.10.2 pointer

Pointer variables are two long.

3.10.3 int, short

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

A negative value is stored in two's complement format. A -2 stored at location 100 would look like:

location	*contents in hex*
100	FE
101	FF

3.10.4 long

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

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

3.10.5 float

A *float* variable is represented internally by a sign flag, a base-256 exponent in excess-64 notation, and a three-character, base-256 fraction. All variables are normalized.

The variable is stored in a sequence of four bytes. The most significant bit of byte 0 contains the sign flag; 0 means it's positive, 1 negative.

The remaining seven bits of byte 0 contain the excess-64 exponent.

Bytes 1,2, and 3 contain the three-character mantissa, with the most significant character in byte 1 and the least in byte 3. The 'decimal point' is to the left of the most significant byte.

As an example, the internal representation of decimal 1.0 is 41 01 00 00.

3.10.6 double

A floating point number of type *double* is represented internally by a sign flag, a base-256 exponent in excess-64 notation, and a seven-character, base-256 fraction.

The variable is stored in a sequence of eight bytes. The most significant bit of byte 0 contains the sign flag; 0 means positive, 1 negative.

The excess-64 exponent is stored in the remaining seven bits of byte 0.

The seven-character, base-256 mantissa is stored in bytes 1 through 7, with the most significant character in byte 1, and the least in byte 7. The "decimal point" is to the left of the most significant character.

As an example, (256**3)*(1/256 + 2/256**2) is represented by the following bytes: 43 01 02 00 00 00 00 00.

For accuracy, floating point operations are performed using mantissas which are 16 characters long. Before the value is returned to the user, it is rounded.

3.11 Floating Point Exceptions

When a C program requests that a floating point arithmetic operation be performed, a call will be made to functions in the floating point support software.

While performing the operation, these functions check for the occurence of the floating point exception conditions; namely, overflow, underflow, and division by zero. On return to the caller, the global integer *flterr* indicates whether an exception has occurred:

flterr	*value returned*	*meaning*
0	computed value	no error has occurred
1	+/- 2.9e-157	underflow
2	+/- 5.2e151	overflow
3	+/- 5.2e151	division by zero

If the value of *flterr* is zero, no error occurred, and the value returned is the computed value of the operation. Otherwise, an error has occurred, and the value returned is arbitrary. The table lists the possible settings of *flterr*, and for each setting, the associated value returned and the meaning.

When a floating point exception occurs, in addition to returning an indicator in *flterr*, the floating point support routines will either log an error message to the console or call a user-specified function. The error message logged by the support routines define the type of error that has occurred (overflow, underflow, or division by zero) and the address, in hex, of the instruction in the user's program which follows the call to the support routines.

Following the error message or call to a user function, the floating point support routines return to the user's program which called the support routines.

To determine whether to log an error message itself or to call a user's function, the support routines check the first pointer in *Sysvec*, the global array of function pointers. If it contains zero (which it will, unless the user's program explicitly sets it), the support routines log a message; otherwise, the support routines call the function pointed at by this field.

A user's function for handling floating point exceptions can be written in C. The function can be of any type, since the support routines don't use the value returned by the user's function. The function has two parameters: the first, which is of type *int*, is a code identifying the type of exception which has occurred. The value 1 indicates underflow, 2 overflow, and 3 division by zero.

The second parameter passed to the user's exception-handling routine is a pointer to the instruction in the user's program which follows the call instruction to the floating point support routines. One way to use this parameter would be to declare it to be of type *int*. The user's routine could then convert it to a character string for printing in an error message.

The example below demonstrates how floating point errors can be trapped and reported. In *main*, a pointer in the *Sysvec* array is set to the routine, *usertrap*. If a floating point exception occurs during the execution of the program, this routine is called with the arguments described above. The error handling routine prints the appropriate error message, and returns to the floating point support routines.

```
#include <stdio.h>

main() {
    Sysvec[FLT_FAULT] = usertrap;
}

usertrap(errcode,addr)
int errcode,addr;
{
    char buff[4];

    switch (errcode) {
        case '1':
            printf("floating point underflow at %x\n",buff);
            break;
        case '2':
            printf("floating point overflow at %x\n",buff);
            break;
        case '3':
            printf("division by zero at %x\n", buff);
            break;
        default:
            printf("usertrap: invalid code %d \n", errcode);
            break;
    }
```

3.12 Register Variables

A *cc*-compiled program can have up to eight register variables. A *cci*-compiled program can declare variables to be of type *register*, but the compiler will ignore the declaration.

3.13 In-Line Assembly Language Code

Assembly language source can be included in a C program, by surrounding the assembly language code with the preprocessor directives *#asm* and *#endasm*.

When the compiler encounters a *#asm* statement, it copies lines from the C source file to the assembly language file that it's generating, until it finds a *#endasm* statement. The *#asm* and *#endasm* statements are not copied.

While the compiler is copying assembly language source, it doesn't try to process or interpret the lines that it reads. In particular, it won't perform macro substitution.

A program that uses *#asm ...#endasm* must avoid the following placing in-line assembly code immediately following an *if* block; that is, it should avoid the following code:

```
if (...){
   ...
}
#asm
   ...
 #endasm
   ...
```

The code generated by the compiler will test the condition and if false branch to the statement following the *#endasm* instead of to the beginning of the assembly language code. To have the compiler generate code that will branch to the beginning of the assembly language code, you must include a null statement between the end of the *if* block and the *asm* statement:

```
if (...){
   ...
}
;
#asm
   ...
 #endasm
   ...
```

3.14 Writing machine-independent code

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

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

3.14.1 Compatibility Between Aztec Products

Within releases, code can be easily moved from one implementation of Aztec C to another. Where release numbers differ (i.e. 1.06 and 2.0) code is upward compatible, but some changes may be needed to move code down to a lower numbered release. The downward compatibility problems can be eliminated by not using new features of the higher numbered releases.

3.14.2 Sign Extension For Character Variables

If the declaration of a *char* variable doesn't specify whether the variable is signed or unsigned, the code generated for some machines assumes that the variable is signed and others that it's unsigned. For example, none of the 8 bit implementations of Aztec C sign extend characters used in arithmetic computations, whereas all 16 bit implementations do sign extend characters. This incompatibility can be corrected by declaring characters used in arithmetic computations as unsigned, or by AND'ing characters used in arithmetic expressions with 255 (0xff). For instance:

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

3.14.3 The MPU... symbols

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

symbol	*processor*
MPU68000	68000
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502
MPU8080	8080
MPUZ80	Z80
MPUINT	Interpreter

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

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

```
#ifdef MPU68000
   /* 68000 code */
#else
#ifdef MPU8086
   /* 8086 code */
#else
#ifdef MPU8080
   /* 8080 code */
#endif
#endif
#endif
```

4. Error checking

Compiler errors come in two varieties-- fatal and not fatal. 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. Both kinds of errors are described in the *Errors* chapter. The non-fatal sort are introduced below.

The compiler will report any errors it finds in the source file. It will first print out a line of code, followed by a line containing the up-arrow (caret) character. The up-arrow in this line indicates where the compiler was in the source line when it detected the error. The compiler will then display a line containing the following:

* The name of the source file containing the line;
* The number of the line within the file;
* An error code;
* The symbol which caused the error, when appropriate.

The error codes are defined and described in the *Errors* chapter.

The compiler writes error messages to its standard output. Thus, error messages normally go to the console, but they can be associated with another device or file by redirecting standard output in the usual manner. For example,

cc prog	errors sent to the console
cc prog >outerr	errors sent to the file *outerr*

The compiler normally pauses after every fifth error, and sends a message to its standard output asking you want to continue. The compiler will continue only if you enter a line beginning with the character 'y'. If you don't want the compiler to pause in this manner, (if, for example, the compiler's standard output has been redirected to a file) specify the *-B* option when you start the compiler.

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, it is a general rule of thumb that the very first error should be corrected first. This may clear up some of the errors which follow.

The best way to attack an error is first to look up the meaning of the error code in the back of this manual. Some hints are given 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.

THE ASSEMBLERS

Chapter Contents

The Assemblers

as and *asi* are relocating assemblers that translate an assembly language source program into relocatable object code. The two assemblers support different machines: *as* accepts assembly language for a 6502 or 65c02; *asi* accepts assembly language for a "pseudo machine".

In an executable program, an *asi*-assembled module must be interpreted by a routine that is in the Aztec libraries.

An executable program can contain both modules that have been assembled with *as* and modules that have been assembled with *asi.*

This description has three sections: the first describes how to operate the assembler; the second describes the assembler's options; and the third presents information of interest to those writing assembly language programs.

1. Operating Instructions

Operationally, the two assemblers are very similar. In the following paragraphs, we will use the name *as* when referring to features that are common to both assemblers. When the two assemblers differ, we will say so.

as is started with a command line of the form

```
as [-options] prog.asm
```

where *[-options]* are optional parameters and *prog.asm* is the name of the file to be assembled. *as* reads the source code from the specified file, translates it into object code, and writes the object code to another file.

1.1 The Source File

The extension on the source file name is optional. If not specified, it's assumed to be *.asm.* For example, with the following command, the compiler will assume that the file name is *test.asm.*

```
as test
```

as will append *.asm* to the source file name only if it doesn't find a period in the file name. So if the name of the source file really doesn't have an extension, you must compile it like this:

```
as filename.
```

The period tells the assembler not to append *.asm* to the name.

1.2 The Object File

By default, the name of the file to which *as* writes object code is derived from the name of the source code file, by changing its extension to *.o* (or to *.i*, if *asi* is used). Also by default, the object code file is placed in the directory that contains the source code file. For example, the command

```
as test.asm
```

writes object code to the file *test.o* (or to *test.i*, if *asi* is used), placing this file in the current directory.

You can explicitly specify the name of the object code file, using the *-O* option. The name of the object code file follows the *-O*, with spaces between the *-O* and the file name. For example, the following command assembles *test.asm*, writing the object code to the file *prog.out*:

```
as -o prog.out test.asm
```

1.3 The Listing File

The *-L* option causes the assembler to create a file containing a listing of the program being assembled. The file is placed in the directory that contains the object file; its name is derived from that of the object file by changing the extension to *.lst*.

1.4 Searching for *instxt* files

The *instxt* directive tells *as* to suspend assembly of one file and assemble another; when assembly of the second file is completed, assembly of the first continues.

You can make the assembler search for *instxt* files in a sequence of directories, thus allowing source files and *instxt* files to be in different directories.

Directories that are to be searched are defined just as for the compilers; that is, using the -I assembler option and the INCLUDE environment variable. Optionally, the compiler can also search the current directory.

Directory search for a particular *instxt* directive can be disabled by specifying a directory name in the directive. In this case, just the specified directory is searched.

1.4.1 The -I option

A -I option defines a single directory to be searched. The directory name follows the -I, with no intervening blanks. For example, the following *-I* option tells the assembler to search the *include* directory on the *ram* volume:

```
-I/ram/include
```

1.4.2 The INCLUDE environment variable.

The INCLUDE environment variable defines a directory to be searched for *instxt* files. For example, the following command sets *INCLUDE* so that the compiler will search for *instxt* files in the directory */ram/include*:

```
set INCLUDE=/ram/include
```

See the SHELL chapter for details on the setting of environment variables.

1.4.3 The search order

Directories are searched in the following order:

1. If the *instxt* directive delimited the file name with the double quote character, ", the current directory on the default drive is searched. If delimited by angle brackets, < and >, this directory isn't automatically searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directory defined in the INCLUDE environment variable is searched.

2. Assembler Options

The assembler supports the following options:

Option	*Meaning*
-O objname	Send object code to *objname*.
-L	Generate listing.
-C	Disable assembly of 65C02 instructions. Not supported by *asi*.
-ZAP	Delete the source file after assembling it.

3. Programming Information

This section discusses the assembly language that is supported by *as*. A description of the assembly language supported by *asi* is not available.

as supports the standard MOS Technology syntax: a program consists of sequence of statements, each of which is in the standard MOS Tech form; and the assembler supports the MOS Tech mnemonics for the standard instructions. *as* supports some of the MOS Tech directives and their mnemonics; it also supports others, as defined below.

The following paragraphs define in more detail the language supported by *as*.

3.1 Statement Syntax

[label] [opcode] [arguments] [[;]comment]

where the brackets "[...]" indicate an optional element.

3.2 Labels

A statement's label field defines a symbol to the assembler and assigns it a value. If present, the symbol name begins in column one. If a statement is not labeled, then column one must be a blank, tab, or asterisk. An asterisk denotes a comment line.

Normally, the symbol in a label field is assigned as its value the address at which the statement's code will be placed. However, the *equ* directive can be used to create a symbol and assign it some other value, such as a constant.

A label can contain up to 32 characters. Its first character must be an alphabetic character or one of the special characters '_' or '.'. Its other characters can be alphabetic characters, digits, '_', or '.'. A label followed by "#" is declared external.

The *cc* compiler places a '_' character at the end of all labels that it generates.

3.3 Opcodes

The assembler supports the standard MOS Tech instruction mnemonics for both the 6502 and 65C02 processors. The directives it supports are defined below.

3.4 Arguments

A statement's arguments can specify a register, a memory location, or a constant.

A memory location can be referenced using any of the standard 6502 or 65C02 addressing modes, and using the standard MOS Tech syntax.

A memory location reference or a constant can be an expression containing any of the following operators:

*	multiply
/	divide
+	add
-	subtract
#	constant
=	constant
<	low byte of expression
>	high byte of expression

Expressions are evaluated from left to right with no precedence as to operator or parentheses.

3.5 Constants

The default base for numeric constants is decimal. Other bases are specified by the following prefixes or suffixes:

Base	*Prefix*	*Suffix*
2	%	b,B
8	@	o,O,q,Q
10	null,&	null
16	$	h,H

A character constant consists of the character, preceded by a single quote. For example: 'A.

3.6 Directives

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

END

end

The *end* directive defines the end of the source statements.

CSEG

cseg

The *cseg* directive selects a module's code segment: information generated by statements that follow a *cseg* directive is placed in the module's code segment, until another segment-selection directive is encountered.

DSEG

dseg

The *dseg* directive selects a module's data segment: information generated by statements that follow a *dseg* directive is placed in the module's data segment, until another segment-selection directive is encountered.

EQU

symbol *equ* *<expr>*

The *equ* directive creates a symbol named *symbol* (if it doesn't already exist), and assigns it the value of the expression *expr*.

PUBLIC

public *<symbol>[,<symbol>...]*

The *public* directive identifies the specified symbols as having external scope. If a specified symbol was created within the module that's being assembled (by being defined in a statement's label field), this directive allows it to be accessed by other modules. If a symbol was not created within the module that's being assembled, this directive tells the assembler that the symbol was created and made public in another module.

BSS

bss *<symname>,<size>*

The *bss* directive creates a symbol named *symnam* and reserves *size* bytes of space for it in the uninitialized data segment. The symbol cannot be accessed by other modules.

GLOBAL

global *<symnam>,<size>*

The *global* directive creates a symbol named *symnam* that other modules can access using the *global* and *public* directives.

If other modules create *symnam* using just the *global* directives, then *symnam* will be located in a program's uninitialized data area. In this case, the amount of space reserved in this area for *symnam* will equal the largest value specified by the *size* fields in the *global* statements that define *symnam.*

If other modules define *symnam* in a *public* statement, but none of them create *symnam* (by specifying it in a label field), then *symnam* will still be located in the uninitialized data segment and space will be reserved for it as defined above.

If one module both defines *symnam* using a *public* statement and creates the symbol by specifying it in a label field, then *symnam* will be located in the program's code or data segment and no space will be reserved for it in the uninitialized data segment.

ENTRY

entry *<symnam>*

The *entry* directive defines the symbol *symnam* as being a program's entry point.

When a program is linked, the linker normally places a jump instruction at the program's base address. If the linker found a module containing an *entry* directive, it sets the target of the jump to the location that was specified in the last *entry* directive that it found; otherwise, it sets the target to the beginning of the program's code segment.

FCB

[label] *fcb* *<value>[,<value>, <value> ...]*

Each *value* in an *fcb* directive causes one or more bytes of memory to be allocated and then initialized to the specified value. The memory is allocated in the currently active segment (code or data, as defined by the last segment-selection directive).

FDB

[label] *fdb* *<value>[,<value>, <value> ...]*

The *fdb* directive is like *fcb*, except that each *value* causes a two-byte field of memory to be allocated and initialized.

FCC

[label] *fcc* *"string"*

The *fcc* directive allocates a field that has the same number of characters as are in *string*, and places *string* in it. The field is placed in the currently-active segment.

RMB

[label] *rmb* *<expr>*

The *rmb* directive reserves a field containing *expr* bytes in the currently-active segment. The contents of the field are not defined.

INSTXT

instxt *<file>*
instxt *"file"*
instxt */file/*

The *instxt* directive causes the assembler to suspend assembly of the current source file and to assemble the source that's in *file*. When done, the assembler will continue assembling the original file.

The assembler can search for a file in several directories. If *file* is surrounded by quotes or by slashes, the assembler will begin the search at the current directory; it will then search directories specified in the -I option and the INCLUDE environment variable. If *file* is surrounded by <>, the assembler will search just the -I and INCLUDE directories.

THE LINKER

Chapter Contents

The Linker

The *ln* linker has two functions:

* It ties together the pieces of a program which have been compiled and assembled separately;
* It converts the linked pieces to a format which can be loaded and executed.

The pieces must have been created by the Manx assembler.

The first section of this chapter presents a brief introduction to linking and what the linker does. If you have had previous experience with linkage editors, you may wish to continue reading with the second section, entitled "Using the Linker." There you will find a concise description of the command format for the linker.

1. Introduction to linking

Relocatable Object Files

The object code produced by the assembler is "relocatable" because it can be loaded anywhere in memory. One task of the linker is to assign specific addresses to the parts of the program. This tells the operating system where to load the program when it is run.

Linking hello.o

It is very unusual for a C program to consist of a single, self-contained module. Let's consider a simple program which 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 which you might have written, but it already happens to be provided in the file, *c.lib*. This file is a library of all the standard i/o functions. It also contains many support routines which 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* was made, it pulls the function from the library and combines it with the "hello, world" program. The link command would look like this:

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

When *hello.c* was compiled, calls were made to some invisible support functions in the library. So linking without the standard library will cause some unfamiliar symbols to be undefined.

All programs will need to be linked with one of the versions of *c.lib.* Initially, you can use *c.lib* itself. Later on, if you find that *c.lib* doesn't suit your requirements, you can use one of the other versions. For more details, see the Libraries section of the Technical Information chapter.

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 will find two function calls in file 1. The reference to *func1* is "resolved" when the definition of *func1* is found in the same file. The following command

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

will cause an error indicating that *func2* is an undefined symbol. The reason is that the definition of *func2* is in another file, namely *file2.o.* The linkage has to include this file in order to be successful:

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

```
file 1                  file 2

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

func1()
{
  return;
}
```

Libraries

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

All the object files specified to the linker will be "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 which satisfy a previous function call are pulled in.

For 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 that have been referenced but not defined. Global variables and all function names are considered to be global symbols.

The list of undefined's 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 will discover that *printf* is defined in a library module whose name also happens to be *printf.* (There is not any necessary 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 undefined's. The linker will not search the library twice to resolve any references which remain unresolved. A common error lies in the following situation:

module of program	*references (function calls)*
main.o	getinput, do_calc
input.o	gets
calc.o	put_value
output.o	printf

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

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

But it is important that *proglib.lib* is built in the right order. Let's 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().*

What happens at link time if *proglib.lib* is built as follows?

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

After *main.o*, the linker has *getinput* and *do_calc* undefined (as well as some other support functions in *c.lib*). Then it begins the search of *proglib.lib*. It looks at the library module, *input*, first. Since that module defines *getinput*, that symbol is taken off the list of undefined's. 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 are 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 will look like this:

```
Undefined symbol: put_value_
```

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

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

The standard libraries were put together with much care so that this kind of problem would not arise.

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 second time through *proglib.lib*, the linker will pull in the module *output*. The reason this is not the most satisfactory solution is that the linker has to search the library twice; this will lengthen the time needed to link.

2. Using the Linker

The general form of a linkage is as follows:

```
ln [-options] file1.o [file2.o ...] [lib1.lib ...]
```

The linker combines object modules produced by the *as* and/or *asi* assemblers into an executable program. It can search libraries of object modules for functions needed to complete the linkage; including just the needed modules in the executable program. The linker makes just a single pass through a library, so that only forward references within a library will be resolved.

Types of Programs

The linker can create programs having the following types:

* PRG programs, which can only be executed in the SHELL environment;
* BIN programs, which can be executed in either the SHELL or the Basic Interpreter environments;
* SYS programs, which are ProDOS system programs.

By default, the linker creates a PRG program. The +B option makes it create a BIN program, and the +S option makes it create a SYS program. When creating a BIN or SYS program, you will also have to include the startup routine *samain.o* in the program.

For a complete discussion of the different types of programs, see the Command Programs section of the Technical Information chapter.

The executable file

The name of the executable output file can be selected using the -O linker option. If this option isn't used, the linker will derive the name of the output file from that of the first object file listed on the command line, by deleting its extension. In the default case, the executable file will be located in the directory in which the first object file is located. For example,

```
ln prog.o c.lib
```

will produce the file *prog*. The standard library, *c.lib*, will have to be included in most linkages.

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

```
ln -o program mod1.o mod2.o c.lib
```

This command also shows how several individual modules can be linked together. A "module", in this sense, is a section of a program containing a limited number of functions, usually related. These modules are compiled and assembled separately and linked together to produce an executable file.

Libraries

Several libraries of object modules are provided with Aztec C65. The most frequently-used of these are *c.lib*, which contains 6502 versions of the non-floating point functions, and *m.lib*, which contains 6502 versions of the floating point functions. Other libraries are provided with some versions of Aztec C65; for their description, see the Libraries section of the Technical Information chapter.

All programs must be linked with one of the versions of *c.lib*. In addition to containing 6502 versions of all the non-floating point functions described in the Functions chapter, it contains internal functions which are called by compiler-generated code, such as functions to perform long arithmetic.

Programs that perform floating point operations must be linked with one of the versions of *m.lib*, in addition to a version of *c.lib*. The floating point library must be specified on the linker command line before *c.lib*.

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

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

```
ln program.o mylib.lib new.lib m.lib c.lib
```

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

Libraries can be conveniently specified using the *-L* option. For example, the following command is equivalent to the following:

```
ln -o program.o -lmylib -lnew -lm -lc
```

For more information, see the description of the *-L* option in the Options section of this chapter.

3. Linker Options

3.1 Summary of options

3.1.1 General Purpose Options

-O file	Write executable code to the file named *file.*
-Lname	Search the library *name.lib* for needed modules.
-F file	Read command arguments from *file.*
-T	Generate a symbol table file.
-M	Don't issue warning messages.
-N	Don't abort if there are undefined symbols.
-V	Be verbose.

3.1.2 Options for Segment Address Specification

-B addr	Set the program's base address to the hex value *addr.*
-C addr	Set the starting address of the program's code segment to the hex value *addr.*
-D addr	Set the starting address of the program's data segment to the hex value *addr.*
-U addr	Set the starting offset of the program's uninitialized data segment to the hex value *addr.*

3.1.3 Options for Overlay Usage

-R	Create a symbol table to be used when linking overlays.
+C size	Reserve *size* bytes at end of the program's code segment (the overlay's code segment is loaded here). *size* is a hex value.
+D size	Reserve *size* bytes at end of the program's initialized and uninitialized data segments (the overlay's data is loaded here). *size* is a hex value.

3.1.4 Special Options for ProDOS

+B	Create a BIN program.
+S	Create a SYS program.
+H start,end	Define a hole in the program, whose beginning and ending addresses are the hex values *start* and *end.*

3.2 Detailed description of the options

3.2.1 General Purpose Options:

The -O option

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

```
ln -o progout prog.o c.lib
```

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

The -L option

The *-L* option provides a convenient means of specifying to the linker a library that it should search, when the extension of the library is *.lib*.

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

```
ln prog.o -lsubs -lio -lm -lc
```

CLIB is set using the SHELL's *set* command. For example, the following command defines *CLIB* when the libraries are in the directory */ln/libs*:

```
set CLIB=/ln/libs/
```

Note the terminating slash on the *CLIB* variable: this is required since the linker simply prepends the value of the *CLIB* variable to the *-L* string.

The -F option

-F file causes the linker to merge the contents of the given file with the command line arguments. For example, the following command causes the linker to create an executable program in the file *myprog*. The linker includes the modules *myprog.o*, *mod1.o*, and *mod2.o* in the program, and searches the libraries *mylib.lib* and *c.lib* for needed modules.

```
ln myprog.o -f argfil c.lib
```

where the file *argfil*, contains the following:

```
        mod1.o mod2.o
        mylib.lib
```

The linker arguments in *argfile* can be separated by tabs, spaces, or newlines.

There are several uses for the *-F* option. The most obvious is to supply the names of modules that are frequently linked together. Since all the modules named are automatically pulled into the linkage, the linker does not spend any time in searching, as with a library. Furthermore, any linker option except *-F* can be given in a *-F* file. *-F* can appear on the command line more than once, and in any order. The arguments are processed in the order in which they are read, as always.

The -T option

The *-T* option creates a disk file which contains a symbol table for the linkage. 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 Memory Organization section of the Technical Information chapter.

The -M option

The linker issues the message "multiply defined symbol" when it finds a symbol that is defined with the assembly language directives *global* or *public* in more than one module. The *-M* option causes the linker to suppress this message unless the symbol is defined in more than one *public* directive.

To maintain compatibility with previous versions of Aztec C, the linker will generate code for a variable that is defined in multiple *global* statements and in at most one *public* statement, and also issue the "multiply defined symbol" message. Thus, if you use the *global* and *public* directives in this way, and don't want to get this message, use the *-M* option to suppress them.

The definition of a symbol in more than one *public* directive is never valid, so the *-M* option doesn't suppress messages in this case.

For more information, see the discussion on global symbols in the Programmer Information sections of the Compiler and Assembler chapters.

The -N option

Normally, the linker halts without generating an executable program if there are undefined symbols; The *-N* option causes the linker to go ahead and generate an executable program anyway.

The -V option

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.

3.2.2 Options for segment address specification

The linker organizes a program into three segments: code, initialized data, and uninitialized data areas. You can define the starting address of these segments using the -C, -D, and -U linker options, respectively. A fourth linker option, -B, will set the "base address" of the program. These options are followed by the desired offset, in hex.

By default, the base address of a PRG or BIN program is 0x800, while the base address of a SYS program is 0x2000. Also by default, a program's code segment starts three bytes after the base address, its initialized data segment follows the code, and its uninitialized data follows the initialized data.

A file created by the linker contains a memory image of the program, from its base address through the end of its code or initialized data segments (whichever is higher). This image is loaded into memory, with the first byte in the file loaded at the program's base address.

By default, a program is expected to begin execution at its base address. Most programs have a startup routine, which performs initialization activities and then calls the program's *main* function. This entry point to the startup routine is usually somewhere in the middle of the program, so at the base address the linker will normally set a jump instruction to the entry point.

You can explicitly specify that a label in a module is an entry point by placing the label in the operand field of the module's assembly language *entry* directive. For example, the *crt0* module in *c.lib* contains the function *.begin.* This label is declared in a *public* directive and also in the module's *entry* directive. When a C module is compiled, the compiler always generates a reference to *.begin*; thus, when the program is linked, *ln* will include the *crt0* module from *c.lib* and place a jump to *.begin* at the program's base address.

If the linker doesn't find a startup routine when it links a program, it won't set the jump instruction at the program's base address. In this case, if you don't specify a starting offset for the program's code segment, it will begin right at the base address.

For example, the following command sets the base address of *prog* to 0x4000:

```
ln -b 4000 -o prog prog.o -lc
```

Because none of the other segment selection options were used in this example, the program's code will begin at offset 0x4003, followed by its initialized data, followed by its uninitialized data.

In the next example, the program's base address is set to 0x900 the offset of its code, initialized data, and uninitialized data segments to 0x2000, 0x2800, and 0x3000, respectively:

```
ln -b 900 -c 2000 -d 2800 -u 3000 prog.o -lc
```

3.2.3 Options for Overlay Usage

The *-R* option causes the linker to generate a file containing the symbol table. It's used when linking a program which calls overlays.

The name of the symbol table file is derived from that of the executable file by changing the extension to *.rsm*. The file is placed in the same directory as the executable file.

The linker reserves space in a program between its uninitialized data area and its heap, into which the program's overlays will be loaded. The amount of space equals the sum of the values that you define using the *+C* and *+D* options. For example,

```
ln +c 3000 +d 1000 prog.o -lc
```

will reserve 0x4000 bytes for overlays. See the Overlay section of the Technical Information chapter for more details.

3.2.4 Special Options for ProDOS

The +B Option

The *+B* option causes the linker to set the type of a file containing a created program to BIN.

The +S Option

The *+S* option causes the linker to set the type of a file containing a created program to SYS and to set the default base address for the program to 0x2000.

The +H Option

The *+H* option defines a "hole"; that is,an area of memory into which the linker should not place a program's code or data. You can create at most four holes in a program using *+H* options.

The option has the following form:

```
        +h start,end
```

where *start* and *end* are the addresses, in hex, of the hole's starting and ending addresses.

For example, suppose you want to create a program, *line*, that uses the primary graphics page (between addresses 0x2000-0x4000) and that begins at address 0x800. The following command will link the program:

```
        ln +h 2000,4000 line.o -lc
```

The linker will place as much of the program's code and data as possible in the area between 0x800-0x2000, and place any additional code and data in the area above 0x4000.

The linker creates a program's code segment by concatenating module code segments, until and unless a module's code overlaps a reserved area. If this occurs, the linker moves the module's entire code segment above the reserved area, in the first non-reserved area in which it will entirely fit, and then continues the concatenation of module code segments.

The linker creates a program's initialized data segment in the same way: it concatenates module initialized data segments as much as possible, without overlapping a reserved area and without breaking a module's initialized data segment into discontiguous pieces.

Because the linker won't break up a module's code segment or data segment, it's likely that some space below a hole will be left unused by the linker.

UTILITY PROGRAMS

Chapter Contents

Utility Programs

This chapter describes commands whose code is built into the SHELL and utility programs that are provided with this package.

NAME

arcv & mkarcv - source dearchiver & archiver

SYNOPSIS

arcv arcfile

mkarcv arcfile

DESCRIPTION

arcv extracts the source from the archive *arcfile*, which has been previously created by *mkarcv*, placing the results in separate files in the current directory.

mkarcv creates the archive file *arcfile*, placing in it the files whose names it reads from its standard input. Only one file name is read from a standard input line.

EXAMPLES

For example, the file *header.arc* contains the source for all the header files. To create these header files, enter:

```
arcv header.arc
```

The files will be created in the current directory.

The following command creates the archive *myarc.arc* containing the files *in.c*, *out.c*, and *hello.c*:

```
mkarcv myarc.arc <myarc.bld
```

The names of the three files are contained in the file *myarc.bld*:

```
in.c
out.c
hello.c
```

NAME

bye - exit to monitor

SYNOPSIS

bye

DESCRIPTION

bye transfers control of the processor to the Apple monitor program that's in ROM, by jumping to location $FF65. To return to the SHELL from the monitor, enter the command

```
3D0G
```

On machines having the autorestart ROM, you can also reenter the SHELL by hitting the *reset* key.

NAME

cat - catenate and print

SYNOPSIS

cat [file] [file] ...

DESCRIPTION

cat reads each *file* in sequence and writes it to its standard output device. If no files are specified, *cat* reads from its standard input device.

Each argument can specify a complete or partial file name, in the normal manner.

By default, *cat*'s standard input and output devices are assigned to the console. Either or both can also be redirected to another device or file, if desired, in the normal fashion.

The code for *cat* is built into the SHELL.

EXAMPLES

```
cat hello.c
```

Writes *hello.c* to the screen.

```
cat hello.c input.c >cat.out
```

Writes *data:/hello.c* and *input.c*, in that order, to *cat.out.*

```
cat
```

Copies typed characters to the screen.

```
cat >../newfile
```

Copies typed characters to *../newfile.*

```
cat </stdio/printf.c >tmp.c
```

Equivalent to *cat /stdio/printf.c >tmp.c.*

SEE ALSO

cp

NAME

cd - change current directory

SYNOPSIS

cd [directory]

DESCRIPTION

cd makes *directory* the current directory. If *directory* isn't specified, the current directory is set to the directory that's defined in the HOME environment variable, if this variable exists.

If the specified directory doesn't exist, the current directory is unchanged.

The *directory* argument defines the path of directories which must be passed through to reach the new current directory. The path can define a complete path from the root directory or it can define a partial path, which is assumed to begin at the current directory.

The code for *cd* is contained in the SHELL.

EXAMPLES

```
cd /work/io
```

The directory */work/io* is made the current directory.

```
cd subs/io
```

The directory *io*, which is reached from the current directory by passing through the subdirectory *subs* of the current directory and then into *io*, is made the current directory. For example, if */work* was the current directory, then after this command */work/subs/io* is the new current directory.

```
cd ..
```

The current directory is set to the parent directory of the directory which was current before the issuance of this command.

```
cd ../include
```

The current directory is set to the directory which is reached by passing through the parent directory of the directory which was current before the issuance of this command and then to its *include* subdirectory.

```
cd ../..
```

The current directory is set to the directory which is reached by passing through the parent directory of the directory which was current before the issuance of this command and then to its parent directory.

NAME

cmp - File comparison utility

SYNOPSIS

cmp [-l] file1 file2

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 the *-l* option isn't specified, the program will stop after the first difference, displaying a message in the format:

```
Files differ: character 10
```

If the *-l* option is specified, *cmp* will list all differences, in the format:

```
decimal-offset  hex-offset  file1-valuefile2-value
```

EXAMPLES

```
cmp otst ntst

Files differ: character 10
```

and

```
cmp -l otst ntst

 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 *as* or *asi* assemblers, libraries of object modules created by the *lb* librarian, and 'rsm' files created by the *ln* linker during the linking of an overlay root.

For example, the following displays the size and symbols for the object module *sub1.o*, the library *c.lib*, and the rsm file *root.rsm*:

```
cnm sub1.o c.lib root.rsm
```

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 sub1.o > dispfile
```

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 aren't printed;
* *cc* is the number of bytes in the module's code segment, in decimal;
* *dd* is the number of bytes in the module's initialized data segment, in decimal;
* *uu* is the number of bytes in the module's uninitialized data segment, in decimal;
* *tt* is the total number of bytes in the module's three segments, in decimal;
* *hh* is the total number of bytes in the module's three segments, in hexadecimal.

If *cnm* displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the modules' 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.

The *-s* option tells *cnm* to display just the sizes of the object modules. If this option isn't specified, *cnm* also displays information about each named symbol in the object modules.

When *cnm* displays information about the modules' named symbols, the *-l* option 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 isn't used, *cnm* displays the information about several symbols on a line and only displays the first eight characters of a symbol's name.

The *-o* option 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 isn't specified, this information is listed just once for each module: prefixed to the first line generated for the module.

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

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

cnm displays information about an module's 'named' symbols; that is, about the symbols that begin with something other than a dollar sign 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.

If the first character of a symbol's type code is lower case, the symbol can only be accessed by the module; that is, it's local to the module. If this character is upper case, 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, which 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 EQUATE directive. The value listed is the equated value of its symbol. The compiler doesn't 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 which are declared outside any function. Static variables are local to the program and so have type *dt*; all other variables are global, that is, accessable from other programs, and hence have type *Dt*.

ov When an overlay is being linked and that overlay itself calls another overlay, this type of symbol can appear in the rsm file for the overlay that is being linked. It indicates that the symbol is defined in the program that is going to call the overlay that is being linked.

The value is the offset of the symbol from the beginning of the physical segment that contains it.

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 didn't create the symbol in a statement's label field.

The compiler generates *Un* symbols for functions that are called but not defined within the program, for variables that are declared to be *extern* and that are actually used within the program, and for uninitialized, global dimensionless arrays. Variables which are declared to be *extern* but which are not used within the program aren't mentioned in the assembly language source file generated by the compiler and hence don't appear in the object file.

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

The compiler generates *bs* symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

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

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

The compiler generates *Gl* symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *Gl* symbols for variables declared using the *global* directive which have a non-zero size.

NAME

config - Define device attributes

SYNOPSIS

config [file]

DESCRIPTION

config defines device attributes to a program that was created using Aztec C. It does this by modifying, as directed by you, the table that defines these attributes.

This description of *config* first describes the device table, then how to use *config*, and finally gives some examples of *config* usage.

1. The Device Table

The device table used by the SHELL and by programs of type PRG (ie, programs that can only be run in the SHELL environment, because they have been linked with the *shmain* module instead of the *samain* module), resides in the SHELL's memory-resident environment area. This area, which contains information that the SHELL needs to maintain between program executions, is loaded from the file that contains the SHELL. It is loaded only when necessary: when the SHELL is first loaded and when the SHELL detects that its environment area has been corrupted. PRG programs and BIN programs created using Aztec C won't corrupt the SHELL's environment area unless they go out of their way to do so, and the SHELL tries to keep programs that were created without using Aztec C from corrupting this area by setting the HIMEM field to the base of this area. So when you're doing standard development activities under the SHELL, the SHELL's memory-resident environment area (and hence the SHELL's memory-resident device table) is normally only loaded once: when the SHELL is first loaded.

A program created using Aztec C that either runs under ProDOS and is of type BIN and SYS (ie, that can run outside the SHELL environment) or that runs under DOS 3.3 uses a device table that resides in the program's memory space. This table is loaded along with the program from a copy in the file from which the program is loaded.

config can modify the copy of a program's device table that is in the file that contains the program. For example, the SHELL is just a program created using Aztec C, and you can modify the device table that is within the file that contains the SHELL, *shell.system*. The modifications that you make to this copy of the SHELL's device table will take affect (ie, affect executing SHELL and PRG-type programs) when the SHELL next reloads this table into its memory-resident environment area.

config can also directly modify the SHELL's memory-resident device table, without modifying a file-resident device table. This feature allows you to temporarily redefine device attributes; you can modify a program's file-resident table when you want to make a more permanent redefinition of device attributes.

Before getting into the description on *config* usage, we are going to define the names by which you will refer to devices. Then, since *config* lets you access a device table at a low level, allowing you to examine and modify specific fields and bits, we define the device table in detail.

1.1 Device Names

The following list defines the names of devices:

con:	The console device
pr:	The printer
ser:	The serial device
sx:	The device that's in slot *x*. For example *s3:* is the name of the device in slot 3.

1.2 Device Table Organization

The include file *device.h* contains definitions related to devices. This section defines information that is in that file.

A device table has the following structure:

```
struct _dev_info {
     char            fnd_str[14];    /* signature */
     short           con_flags;      /* console flags */
     struct sgttyb   tty;            /* con info for ioctl */
     struct _name_dev
                     dev_con,        /* con: info */
                     dev_pr,         /* pr: info */
                     dev_ser;        /* ser: info */
     struct _slot_dev
                     slots[8];       /* slot info */
     int             init_max;       /* init space size */
     int             init_len;       /* init space used */
     char            init_buf[];     /* init space */
};
```

1.2.1 The *fnd_str* Field

The *fnd_str* field contains a (hopefully) unique character string; when told to find a file's device table, *config* looks for this string.

1.2.2 The *con_flags* Field

The *con_flags* field contains bits that define console attributes. the following tables lists for each bit its symbolic name, a value in parentheses that defines its location in the field, and the meaning of the bit.

CON_IMAP (0x01)
: When this bit is set, the console driver will perform keyboard mapping. This mapping is needed needed for Apples whose keyboards don't support the full ASCII character set.

CON_UPPR (0x02)
: When this bit is set, the console driver will turn lower case characters into upper on output.

CON_HIGH (0x80)
: When this bit is set, the console driver will set the high order bit of each character it outputs.

The *config* commands *dump con:* and *mod con:* are used to display and modify the *con_flags* field.

1.2.3 The *tty* Field

The *tty* field contains those fields related to a program's console I/O that the program modifies by calling the *ioctl* function. For example, this field defines whether the console is in line-oriented or character-oriented mode. *config* doesn't modify this field, so we won't discuss it further here. For a complete discussion of console I/O and *ioctl*, see the *Console I/O* section of the Library Overview chapter.

1.2.4 The *dev_con*, *dev_pr*, and *dev_ser* Fields

The *dev_con*, *dev_pr*, and *dev_ser* fields define the slot devices that are associated with the *con:*, *pr:*, and *ser:* devices, respectively. These three devices are just generic names that allow a program to access the console, printer or serial card without having to know the exact slot that contains the card. When a program performs I/O to one of these generic devices, the I/O operation is simply passed on to its associated slot device. For example, if a program writes to *pr:*, and during one execution the device table used by the program specifies that *pr:* is associated with slot device *s1:*, then the data will be written to the card in slot 1. If during another execution the table specifies that *pr:* is associated with slot device *s2:*, then the data will be written to the card in slot 2.

The *config* commands *dump con:*, *dump pr:*, and *dump ser:* display the slot device that's associated with a generic device. And the commands *mod con:*, *mod pr:*, and *mod ser:* define the slot device that's associated with a generic device.

The console device driver does not currently look at the *dev_con* field; it always performs console input by calling the RDKEY ROM routine that begins at 0xfd0c, which in turn calls the console input routine whose address is in the KSW zero-page field. It performs console output by calling the COUT ROM routine that begins at 0xfded, which in turn calls the console output routine whose address is in the CSW zero-page field. If the Apple on which the SHELL is running has an 80 column card, the SHELL will automatically set up the CSW and KSW zero page fields to use it.

1.2.5 The *slot* Fields

The *slot* field is an array of eight structures of type *struct __slot_dev*, each of which defines the attributes of one slot device. This structure will be defined after we finish describing the fields in the device table structure.

1.2.6 The *init_buf*, *init_max*, and *init_len* Fields

When a program opens a slot device, the slot device driver can optionally send an initialization string to the device. The following *__dev_info* fields define information related to device initialization strings:

* *init_buf* contains the strings;
* *init_max* defines the length of the *init_buf* space;
* *init_len* defines the number of bytes in *init_buf* that have been allocated to initialization strings.

config allocates an initialization string by placing the string at the current location defined by *init_len* and then incrementing *init_len* past the string.

The size of the SHELL's *init_buf* area is hard-coded into the SHELL to be 64 bytes; this limit can't be exceeded. The size of a BIN, SYS, or DOS 3.3 program's *init_buf* area is also hard-coded to be 64 bytes, but you can change this by modifying and replacing the *devtab* module in the various versions of *c.lib* (*c.lib*, *ci.lib*, *d.lib*, and *di.lib*).

The following *config* commands access the string initialization fields:

* The *dump init* command displays the contents of the *init_max* and *init_len* fields.
* The *mod init* command resets the *init_len* field and turns off each slot device's INIT_STR bit, thus freeing all of the string initialization space.
* The commands that display and modify information about slot devices (*dump sx:* and *mod sx:*) access the device table's string initialization fields.

1.3 The *_slot_dev* Structure

A slot device's *struct __slot_dev* structure has the following form:

```
struct _slot_dev {
        short   outvec;     /* CSW vector ($36-37) */
        short   invec;      /* KSW vector ($38-39) */
        short   init;       /* init str offset in initbuf */
        char    slot;       /* $s0 */
        char    hi_slot;    /* slot number */
        char    type;
                  /* -1=BASIC, 0=Pascal1.0, 1=Pascal1.1 */
        char    flags;      /* slot attr flags*/
        char    tabp;       /* line pos for tab mapping */
        char    tabw;       /* tab width */
        char    iflags;     /* slot init flags */
        char    xtra;       /* unused */
};
```

1.3.1 The *outvec* and *invec* Fields

The *outvec* and *invec* fields in a slot device's *_slot_dev* structure contains the addresses of the routines that the slot device driver will call to transfer each character to or from the slot device, respectively. Normally, the slot device driver sets these addresses to locations within the associated card's ROM space at the time the device is opened, by determining the type of protocol (Basic, Pascal 1.0, or Pascal 1.1) used by the ROM routines. As discussed below, the setting of these fields for a particular slot device by the slot device driver can be disabled by turning off its INIT_VEC bit in the *iflags* field of its *_slot_dev* structure.

config doesn't have commands to modify these fields, although the *dump sx:* commands display their contents.

1.3.2 The *init* Field

The *init* field in a slot device's *_slot_dev* structure contains the offset within the *_dev_info* structure's *init_buf* area at which the device's initialization string begins.

As discussed below, when a slot device is opened, the INIT_STR bit in the *iflags* field of the device's *_slot_dev* structure determines whether the device's initialization string should be sent to it.

The *dump sx:* command will display the initialization string that's associated with slot device *sx:*, and the *mod sx: str=...* command will assign an initialization string to *sx:*.

1.3.3 The *type* Field

The *type* field in a slot device's *_slot_dev* structure defines the type of protocol used by the device's ROM routines. It can have the following values:

Value	*Meaning*
-1	Basic

0	Pascal 1.0
1	Pascal 1.1

This field is normally set when the device is opened, but if the INIT_VEC bit in the *iflags* field of the device's structure is off, the *type* field is not set.

The *dump sx:* command will display *sx:*'s *type* field, but *config* doesn't have a command to set it.

1.3.4 The *slot* and *hi_slot* fields

The *slot* field in a slot device's *_slot_dev* structure defines the high-order byte of the address of the device's ROM space. The *hi_slot* field defines the number of the slot.

The *dump sx:* command will display *sx:*'s *slot* and *hi_slot* fields, but *config* doesn't have commands that will modify these fields.

1.3.5 The *flags* Field

The *flags* field in a slot device's *_slot_dev* structure contains bits defining various attributes of the slot. The following table lists for each bit the symbolic name of the bit, a value in parentheses that defines the location of the bit, and the bit's meaning.

SLOT_LFCR (0x01)
: When this bit is set, an carriage return character that is read from the device will be translated to a linefeed, and a linefeed character that is written to the device will be translated to a carriage return.

SLOT_TABS (0x02)
: When this bit is set, an output tab character will be replaced by enough space characters to position the device at the next "tab stop". For more information on tab stops, see the descriptions of the *tabw* and *tabp* fields.

SLOT_UPPR (0x04)
: When this bit is set, an output alphabetic character will be converted to upper case.

SLOT_CRLF (0x08)
: When this bit is set, and a program writes a carriage return character to the device, the device driver will also write a line feed character to the device.

SLOT_HIGH (0x80)
: When this bit is set, the high order bit will be set on all output characters.

The *dump sx:* command will display in binary the contents of *sx:*'s *flags* field, and the *mod sx:* command can be used to modify it.

1.3.6 The *iflags* Field

The *iflags* field in a slot device's *__slot__dev* structure contains bits that define attributes of the device. The following table lists for each bit the symbolic name of the bit, a value in parentheses that defines the location of the bit, and the bit's meaning.

INIT__VEC (0x01)
: When this bit is set and the slot device is opened, the addresses of its input and output vectors will be determined and set in the structure's *invec* and *outvec* fields based on the type of protocol used by the card, and the *type* field will be set.

INIT__CAL (0x02)
: When this bit is set and the slot device is opened, the card will be initialialized by issuing a call to the first byte of its ROM space.

INIT__STR (0x04)
: When this bit is set and the slot device is opened, the device's initialization string (which is pointed at by the *init* field of the device's *__slot__dev* structure) will be written to the device.

INIT__ONCE (0x08)
: When this bit is set, the initialization activities described above will only be performed once for the slot device.

The *dump sx:* command will display in binary the contents of *sx:*'s *iflags* field, and the *mod sx:* command can be used to modify it.

1.3.7 The *tabw* Field

The *tabw* field in a slot device's *__slot__dev* structure defines the number of characters between each of the device's tab stops. If 0, it's assumed to be eight.

When a tab character is written to a slot device, the slot device driver can optionally output spaces in its place until the next tab stop is reached. This replacement is enabled by setting the SLOT__TABS bit in the *flags* fields of the device's *__slot__dev* structure.

The *dump sx:* and *mod sx:* commands can be used to display and modify the contents of *sx:*'s *tabw* and *tabp* fields.

1.3.8 The *tabp* Field

The *tabp* field in a slot device's *__slot__dev* structure defines the number of characters that have been send to it since the last carriage return or linefeed. When a tab character is sent to a device, the device driver uses the device's *tabp* field to decide how many spaces it should output to reach the next tab stop.

1.3.9 The *xtra* Field

This field is unused.

2. Using *config*

config is an interactive program: you enter commands to it to examine device information, make modifications, and so on. These commands access a copy of the table that resides in an internal buffer within *config*: when you define the file whose device table you want to modify (or the table that's in the SHELL's memory-resident environment area), *config* finds the table and reads it into its internal buffer; your examination and modification commands then access the table that is in this internal buffer. When you've completed the modifications, you type the *write* command, which causes *config* to write the table back to the location from which it was obtained.

The location of a table to be modified can be specified to *config* as an argument when it is started. This argument can be the name of a file containing a program whose device table is to be modified. It can also be the word *mem:*, which specified that the SHELL's memory-resident device table is to be modified. When *config* is started with this optional argument, it automatically searches for the table in the specified location and, if found, reads it into its table; if *config* doesn't find the table, it will tell you.

Alternatively, the location of a device information table that's to be modified can be specified once *config* is active, by typing the *open* command. This command takes a single argument: the name of the file whose device table is to be modified; or *mem:*, if the SHELL's memory-resident table is to be modified. The *open* command searches for the table in the specified location, but doesn't load it into *config*'s internal buffer; if you want the table read, you must explicitly say so, using *config*'s *read* command. This allows you to make changes to one program's device table and then write the modified table to several different programs.

3. Commands

config has just a few basic commands, most of which have arguments. We'll first list the commands and give a brief description. The following paragraphs will then discuss the commands in detail.

command	*Description*
dump	Display information
mod	Make modifications
open	Prepare to examine/modify a device table
read	Read a device table into *config*'s internal buffer
write	Write a device table from *config*'s internal buffer
quit	Halt *config*

3.1 The *dump* Command

The *dump* command displays information about one device, about all devices, or about the device initialization string space. The command has the following format:

```
dump [dev]
```

The optional argument *dev* is the name of the device about which you want to get information, or the word *init* if you want information about string space. If an argument isn't specified, information about all devices and about string space is displayed.

3.2 The *mod* Command

The *mod* command is used to modify device attributes and to reset the device initialization string space. The command has the following format:

```
mod dev [args]
```

where *dev* is the name of the device whose attributes are to be changed, or *init* if string space is to be initialized; *[args]* are arguments defining the attributes that are to be changed. The arguments to the *mod* command depend on the device being modified. The following paragraphs discuss the modification of each device.

3.2.1 Modifying *con:*

The command for modifying *con:* has the following format:

```
mod con: [flags=xx] [imap] [uppr] [high]
```

where square brackets surround optional arguments. *imap*, *uppr*, and *high* usually cause a bit to be turned on in the *con_flags* field. If preceded by a ~ character, they cause the designated bit to instead be turned off.

The arguments have the following meanings.

Argument	*Meaning*
flags=xx	sets the field in the device table that defines console attributes, *con_flags*, to the hex value *xx*.
imap	Set (or reset, if preceded by ~) the CON_IMAP bit in *con_flags*.
uppr	Set or reset the CON_UPPR bit in *con_flags*.
high	Set or reset the CON_HIGH bit in *con_flags*.

3.2.2 Modifying *pr:* and *ser:*

The commands that modify the *pr:* and *ser:* attributes are similar:

```
        mod pr: sx:
        mod ser: sx:
```

where *sx:* is the name of the slot device that is to be associated with *pr:* or *ser:*.

3.2.3 Modifying Slot Devices

The command for modifying the attributes of a slot divice has the following form:

```
        mod sx: [flags=xx] [iflags=xx] [tabw=dd]
             [lfcr] [crlf] [tabs] [uppr] [high] [cal] [once] [vec]
             [str=string]
```

where *sx:* is the name of the slot device.

3.2.3.1 The *flags* and *iflags* Arguments

The *flags=xx* argument sets the device's *flags* field to the hex value *xx*.

Similarly, the *iflags=xx* argument set the device's *iflags* field to the hex value *xx*.

3.2.3.2 The *lfcr*, ... Arguments

The arguments specified on the second line (*lfcr*, ...) usually cause a bit in the device's *flags* or *iflags* field to be turned on; if preceded by a ~ character, they instead cause the designated bit to be reset. The symbolic name of the bit represented by these arguments can be derived by appending "SLOT_" (for a *flags* bit) or "INIT_" (for an *iflags* bit). For example, the command "mod s2: lfcr" sets the bit SLOT_LFCR in the *flags* field in the *_slot_dev* structure for the *s2:* device.

3.2.3.3 The *str=string* Argument

The argument *str=string* usually sets the initialization string for the specified slot device to *string* and turns on the INIT_STR bit for the device. If the argument has the form *~str* (ie, preceded by a ~ character and not followed by *=string*), the INIT_STR bit is instead turned off.

Strings are usually specified by surrounding them with double-quote characters, although if a string contains just printable characters with no spaces it can be specified without the surrounding quotes.

In a quoted string, a printable character is represented by itself. Unprintable characters are represented by a sequence that begins with a backslash character, as defined in the following table.

Sequence	*Meaning*
\n	Newline

\t	Horizontal tab
\b	Backspace
\r	Carriage return
\f	Form Feed
\\	Backslash
\xyy	The hex value *yy*

For example, the command to set the initialization string for slot device *s2:* to a string consisting of an escape character (hex value 1b), followed by the character Q would be:

```
mod s2: str="\x1bQ"
```

3.2.4 Reseting Device Initialization String Space

The following command resets the use of the space used for device initialization strings:

```
mod init
```

It resets the field that points to the top of allocated string space (*init_len* in the device information structure), and turns off the INIT_STR bit and clears the *init* field for each slot device.

3.3 The *open* Command

The *open* command prepares *config* for accessing a device table that is in a file or in the SHELL's memory-resident environment area. The command has the following form:

```
open file
```

where *file* is the name of the file whose device table is to be accessed, or *mem:* to access the table in the SHELL's memory-resident environment area.

If a file is specified, the *open* command causes *config* to open the file and search for the file's device information table. If the table is not found, *config* will say so.

Note that the *open* command does not read the specified device information table into *config*'s internal buffer; you must explicitly tell *config* to do that, using the *read command.*

3.4 The *read* Command

The *read* command causes *config* to read the device information table from the currently open file or SHELL environment area into *config*'s internal buffer.

Note that when a file or the SHELL's environment area is specified as a command-line argument, *config* automatically reads the device information table into its internal buffer, making it unnecessary to issue a *read* command.

3.5 The *write* Command

The *write command* causes *config* to write the device information table that's in its internal buffer to the currently open file or SHELL environment area.

3.6 The *quit* Command

The *quit* command causes *config* to halt.

4. Examples

In this example, the device table in *shell.system* is modified for use with an Image Writer printer that's connected to a Super Serial card in slot 2. The changes are also written to the SHELL's memory-resident table. First, we get *config* started by entering the following command to the SHELL:

```
config shell.system
```

Once started, *config* finds the device table that's in the file *shell.system* and reads it into *config*'s internal buffer.

We next display the current settings of *s2:*'s flags and fields by entering to *config*:

```
dump s2:
```

We next set and reset flags and fields that define how i/o to *s2:* is to be performed. These changes are made just to the device table that's in *config*'s internal buffer; they won't be made to the device table that's in *shell.system* until the *write* command is issued.

```
mod s2: vec cal ~lfcr tabs tabw=4 str="\x1bQ"
```

The operands to the *mod* command have the following meanings:

* *vec* tells the device driver to determine the addresses of *s2:*'s ROM routines when *s2:* is opened;
* *cal* tells the driver to call the device's initialization code, which begins at the first byte of its ROM, when the device is opened;
* *~lfcr* tells the driver not to send a linefeed after a carriage return;
* *tabs* tells the driver to output spaces in place of tabs;
* *tabw=4* says that there are 4 spaces between tab stops;
* *str="\x1bQ"* tells the driver to send the specified character string to the device when it is opened.

We next display the modified settings of *s2:*'s flags and fields, which are in the device table that's in *config*'s internal buffer:

```
        dump s2:
```

Everything's OK, so we write the modified device table back to *shell.system.*

```
        write
```

The changes that have just been made won't affect an executing SHELL or PRG programs until the SHELL detects that its environment area has been corrupted, or until the SHELL is loaded following system power-up. To make these changes take effect immediately, we need to write them to the SHELL's memory-resident device table. To do this, we first tell *config* that we want to access this memory-resident table by entering:

```
        open mem:
```

This doesn't affect the device table that's in *config*'s internal buffer, so we can immediately issue the following command to overwrite the SHELL's memory-resident table:

```
        write
```

We're all done, so we exit *config* by entering:

```
        quit
```

NAME

cp - copy files

SYNOPSIS

cp [-f] infile outfile

cp [-f] file1 [file2 ...] dir

DESCRIPTION

cp copies files, and their attributes. The first form of the command, as shown above, copies *infile* to *outfile*. The second copies *file1, file2,* ... into the directory named *dir*.

The *-f* option causes *cp* to automatically overwrite any existing files. If this option isn't specified and if a file to be created already exists, *cp* will ask if you want it overwritten.

For example, the following command copies the file *hello.c* that is in the current directory to the file *newfile.c* in the */source* directory.

```
cp hello.c /source/newfile.c
```

The next command copies all ".c" files in the */arc/* directory to the current (ie, the ".") directory:

```
cp /arc/*.c .
```

NAME

crc - Utility for generating the CRC for files

SYNOPSIS

crc file1 file2 ...

DESCRIPTION

crc computes a number, called the CRC, for the specified *files.*

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

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

```
crc *.c
```

NAME

date - display date and time

SYNOPSIS

date

DESCRIPTION

Displays the date and time.

NAME

debug - set debug mode

SYNOPSIS

debug

DESCRIPTION

To debug a SHELL-activated program using the monitor program that's in ROM, first enter the *debug* command, and then enter the command to start the program. The SHELL will load the program, perform i/o redirection and pass arguments to it if necessary; then, when it sees that the *debug* command was entered, it will jump to the monitor.

When you're done debugging, return to the SHELL by entering the command

```
3D0G
```

The *debug* command affects only the next program that the SHELL starts. That is, for each program that you want to debug, you must first enter the *debug* command and then enter the command to start the program.

NAME

df - Display Volume info

SYNOPSIS

df [/vol]

DESCRIPTION

Displays information about disk space utilization.

The optional parameter */path* defines the disk of interest. It can be the name of a file or directory; in this case, information is displayed about the disk that contains the specified file or directory.

/path can also be the single character '/', (ie, specify the SHELL's simulated root directory of the entire file system); in this case, information is displayed about all on-line disks.

If */path* isn't specified, and if the current directory isn't the simulated root directory of the file system, information is displayed about the disk that contains the current directory.

If */path* isn't specified, and if the current directory is the simulated root directory of the file system, information is displayed about all on-line disks.

EXAMPLES

The following command displays information about the disk that contains the current directory:

```
df
```

The next command displays information about the */ram* disk:

```
df /ram
```

The next command displays information about all on-line disks:

```
df /
```

NAME

diff - Source file comparison utility

SYNOPSIS

diff [-b] file1 file2

DESCRIPTION

diff is a program, similar to the UNIX program of the same name, that determines the differences between two files containing text. *file1* and *file2* are the names of the files to be compared.

1. The -b option

The -b option causes *diff* to ignore trailing blanks (spaces and tabs) and to consider strings of blanks to be identical. If this option isn't specified, *diff* considers two lines to be the same only if they match *exactly*.

For example, if file1 contains the the line

```
^abc$
```

(^ and $ stand for "the beginning of the line" and "the end of the line", respectively, and aren't 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 the -b option.

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 the -b option.

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.

2. The conversion list

diff writes, to its standard output, a "conversion list" 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*.

2.1 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.

2.1.1 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 are affected, just 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
```

2.1.2 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++;".

```
53,55c60,61
< if (a=b){
<     d = a;
<     a++;
< }
---
> if (a==b)
>     d = a++;
```

3. 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 the -h option. As with the UNIX *diff* when used with the -h option, 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* doesn't support the options e, f, or h.

Unlike the UNIX *diff*, the Manx version requires that both operands to *diff* be actual files. Because of this, the Manx version of *diff* doesn't support the features of the UNIX version which allows one operand to be a directory name, (to specify a file in that directory having the same name as the other operand), and which allows one operand to be '-' (to specify *diff*'s standard input instead of a file).

NAME

echo - echo arguments

SYNOPSIS

echo [arg] [arg] ...

DESCRIPTION

echo writes its arguments, separated by blanks and terminated by a newline, to its standard output device.

The output thus goes, by default, to the screen. It can also be redirected to another device or file in the normal manner.

NAME

grep - pattern-matching program

SYNOPSIS

grep [-cflnv] pattern [files]

DESCRIPTION

grep is a program, similar to the UNIX program of the same name, that searches files for lines containing a pattern. By default, such lines are written to *grep*'s standard output.

1. Input files

The **files** parameter is a list of files to be searched. If no files are specified, *grep* searches its standard input. Each file name can specify a single file to be searched.

2. Options

The following options are supported:

v	Print all lines that don't match the pattern.
c	Print just the name of each file and the number of matching lines that it contained.
l	Print the names of just the files that contain matching lines.
n	Precede each matching line that's printed by its relative line number within the file that contains it.
f	A character in the pattern will match both its upper and lower case equivalent.

3. Patterns

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

Some patterns match just a single character; others, which match strings, can be constructed from those that match single characters. In the following paragraphs, we'll first describe the patterns that match a single character, and then describe patterns that match strings of characters.

3.1 Matching single characters

The patterns that match a single character are these:

* An ordinary character (that is, one other than the special characters described below) matches itself.
* A period (.) is a pattern that matches any character except newline.

* A non-empty string of characters enclosed in square brackets, [], matches any one character in that string. For example, the pattern

  [ad9@]

  matches any one of the characters *a, d, 9,* or @.

  If, however, the string begins with the caret character (^), the regular expression matches any character except the other enclosed characters and newline. The '^' has this special meaning only if it is the first character of the string. For example, the pattern

  [^ad9@]

  matches any single character *except a, d, 9,* or @.

  The minus character ,-, can be used to indicate a range of consecutive ASCII characters. For example, [0-9] is equivalent to [0123456789].

* A backslash (\) followed by a special character matches the special character itself. The special characters are:

  ., *, [, and \, which are always special, except when they appear in square brackets, [].

  ^ (caret), which is special when it is at the beginning of an entire regular expression (as discussed in 3.4) and when it immediately follows the left of a pair of square brackets.

  $, which is special at the end of an entire regular expression (discussed in 3.4).

3.2 Matching character strings

Patterns can be concatenated. In this case, the resulting pattern matches strings whose substrings match each of the concatenated patterns. For example, the pattern

```
abc
```

matches the string *abc.* This pattern is built from the patterns *a, b,* and *c.* The pattern

```
a.c
```

matches strings containing three characters, whose first and last characters are *a* and *c,* respectively, such as

```
abc
a@c
axc
```

3.3 Matching repeating characters

A pattern can be built by appending an asterick (*) to a pattern that matches a single character. The resulting pattern matches zero or more occurrences of the single-character pattern For example, the pattern

```
a*
```

matches any line containing zero or more *a* characters. And the pattern

```
sub[1-4]*end
```

matches lines containing strings such as

```
subend
sub132132end
```

3.4 Matching strings that begin or end lines

An entire pattern may be constrained to match only character strings that occur at the beginning or the end of a line, by beginning or ending the pattern with the character '^' or '$', respectively. For example, the pattern

```
^main
```

matches the line that begins

```
main
```

but not one that begins

```
the main ...
```

The pattern

```
line$
```

matches the line ending in

```
... the end of the line
```

but not the line ending in

```
a hard-hit line drive.
```

4. Examples

4.1 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 aren't 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
```

won't work.

4.2 The special character '.'

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

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

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

4.3 The backslash character

There are occasions when you want to find the character '.' in a file, and don't 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
```

isn't 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 just 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'.

4.4 The dollar sign and the caret ($ 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
```

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

```
grep ^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$
```

4.5 Using brackets

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 don't begin with a digit, you could enter

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

4.6 Repeated characters

Suppose you want to find all lines in the file *prog.c* that contain strings whose first character is 'e' and whose last character is 'z'. The command

```
grep e.*z prog.c
```

will do that. The 'e' matches an 'e', the '.*' matches zero or more arbitrary characters, and the 'z' matches a 'z'.

5. 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.

5.1 Option differences

* The option -f is supported only by the Manx *grep.*
* The options -b and -s are supported only by the UNIX *grep.*

5.2 Pattern differences

Basically, the patterns accepted by the Manx *grep* are a subset of those accepted by the UNIX *grep.*

* The Manx *grep* doesn't allow a regular expression to be surrounded by '\(' and '\)'.
* The Manx *grep* doesn't accept the construct '\{m\}'.
* The Manx *grep* doesn't allow a right bracket, ']', to be specified within brackets.
* Quoted strings can't be passed to the Manx *grep.* For example, the Manx *grep* won't accept

```
grep 'this is a fine kettle of fish' file.doc
```

NAME

hd - hex dump utility

SYNOPSIS

hd [-r] [+n[.]] file1 [+n[.]] file 2 ...

DESCRIPTION

hd displays the contents of one or more files in hex and ascii to its standard output.

file1, *file2*, ... are the names of the files to be displayed.

+*n* specifies the offset into the file where the display is to start, and defaults to the beginning of the file. If +*n* is followed by a period, *n* is assumed to be a decimal number; otherwise, it's assumed to be hexadecimal. Each file will be displayed beginning at the last specified offset.

EXAMPLES

```
hd +16b oldtest newtest +0 junk
```

Displays the data forks of the files *oldtest* and *newtest*, beginning at offset 0x16b, and of the file named *junk* beginning at its first byte.

```
hd -r +1000. tstfil
```

Displays the contents of the resource fork of *tstfil*, beginning at byte 1000.

NAME

lb - object file librarian

SYNOPSIS

lb library [options] [mod1 mod2 ...]

DESCRIPTION

lb is a program that creates and manipulates libraries of object modules. The modules must have been created by the Manx assembler.

This description of *lb* is divided into three sections: the first describes briefly *lb*'s arguments and options, the second *lb*'s basic features, and the third the rest of *lb*'s features.

1. The arguments to *lb*

1.1 The *library* argument

When started, *lb* acts upon a single library file. The first argument to *lb* (*library*, in the synopsis) is the name of this file. The filename extension for *library* is optional; if not specified, it's assumed to be *.lib.*

1.2 The *options* argument

There are two types of *options* argument: function code options, and qualifier options. These options will be summarized in the following paragraphs, and then described in detail below.

1.2.1 Function code options

When *lb* is started, it performs one function on the specified library, as defined by the *options* argument. The functions that *lb* can perform, and their corresponding option codes, are:

function	*code*
create a library	(no code)
add modules to a library	-a, -i, -b
list library modules	-t
move modules within a library	-m
replace modules	-r
delete modules	-d
extract modules	-x
ensure module uniqueness	-u
help	-h

In the synopsis, the *options* argument is surrounded by square brackets. This indicates that the argument is optional; if a code isn't specified, *lb* assumes that a library is to be created.

1.2.2 Qualifier options

In addition to a function code, the *options* argument can optionally specify a qualifier, that modifies *lb*'s 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 it, any of the following option arguments could be specified:

```
-as
-sa
-a -s
-s -a
```

1.3 The *mod* arguments

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* assumes that it's *.o*.

1.4 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 can't itself contain a *-f filename* argument.

2. Basic features of *lb*

In this section we want to describe the basic features of *lb*. With this knowledge in hand, you can start using *lb*, and then read about the rest of the features of *lb* at your leisure.

The basic things you need to know about *lb*, and which thus are described in this section, 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

Thus, with the information presented in this section you can create libraries and get a list of the modules in libraries. The third section of this description shows you how to modify selected modules within a library.

2.1 Creating 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 doesn't contain a function code, and it's 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 exmpl.lib obj1.o obj2.o
```

Making use of *lb*'s assumptions about file names for which no extension is specified, the following command is equivalent to the above command:

```
lb exmpl 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.

2.1.1 The temporary 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.

2.2 Getting the table of contents for a library

To list the names of the modules in a library, use *lb*'s *-t* option. For example, the following command lists the modules that are in *exmpl.lib*:

```
lb exmpl -t
```

The list will include 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.

2.3 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.

2.4 Order 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 tutorial 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 its description later in this chapter.

2.5 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, *lb*'s *-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 can't be nested.

3. 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.

3.1 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 *lb*'s 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.

3.1.1 Adding modules before an existing module

As an exmple 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* looking like this:

```
obj1   mod1   mod2   obj2   obj3
```

Note that in the *lb* command we didn't need to specify the extension of either the file containing the library to which modules were to be added or the extension of the files containing the modules to be added. *lb* assumed that the extension of the file containing the target library was *.lib*, and that the extension of the other files was *.o*.

As an example of the addition of one library to another, suppose that the library *mylib.lib* contains the modules

```
mod1   mod2   mod3
```

and that the library *exmpl.lib* contains

```
obj1   obj2   obj3
```

Then the command

```
lb -b obj2 mylib.lib
```

adds the modules in *mylib.lib* to *exmpl.lib*, resulting in *exmpl.lib* containing

```
obj1   mod1   mod2   mod3   obj2   obj3
```

Note that in this example, we had to specify the extension of the input file *mylib.lib*. If we hadn't included it, *lb* would have assumed that the file was named *mylib.o*.

3.1.2 Adding modules after an existing module

As an example of adding modules after a specified module, the command

```
lb exmpl -a obj1 mod1 mod2
```

will insert *mod1* and *mod2* after *obj1* in the library *exmpl.lib*. If *exmpl.lib* originally contained

```
obj1   obj2   obj3
```

then after the addition, it contains

```
obj1   mod1   mod2   obj2   obj3
```

3.1.3 Adding modules at the beginning or end of a library

The 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 the *-i* and *-a* options, these options aren't followed by the name of an existing module in the library.

For example, given the library *exmpl.lib* containing

```
objl    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 objl 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

```
objl    obj2    mod1   mod2
```

3.2 Moving modules within a library

Modules which already exist in a library can be easily moved about, using the *move* option, *-m*.

As with the options for adding modules to an existing library, there are several forms of *move* functions:

option	*meaning*
-mb target	move modules before the module *target*
-ma target	move modules after the module *target*
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the *lb* command, the names of the modules to be moved follows 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.

3.2.1 Moving modules before an existing module

As an example of the movement of modules to a position before an existing module in a library, suppose that the library *exmpl.lib* contains

```
objl    obj2    obj3    obj4    obj5    obj6
```

The following command moves *obj3* before *obj2*:

```
lb exmpl -mb obj2 obj3
```

putting the modules in the order:

```
objl    obj3    obj2    obj4    obj5    obj6
```

And, given the library in the original order again, the following command moves *obj6*, *obj2*, and *obj1* before *obj3*:

```
lb exmpl -mb obj3 obj6 obj2 obj1
```

putting the library in the order:

```
obj1    obj2    obj6    obj3    obj4    obj5
```

As an example of the movement of modules to a position after an existing module, suppose that the library *exmpl.lib* is back in its original order. Then the command

```
lb exmpl -ma obj4 obj3 obj2
```

moves *obj3* and *obj2* after *obj4*, resulting in the library

```
obj1    obj4    obj2    obj3    obj5    obj6
```

3.2.2 Moving modules to the beginning or end of a library

The options for moving modules to the beginning or end of a library are *-mb+* and *-ma+*, respectively.

For example, given the library *exmpl.lib* with contents

```
obj1    obj2    obj3    obj4    obj5    obj6
```

the following command will move *obj3* and *obj5* to the beginning of the library:

```
lb exmpl -mb+ obj5 obj3
```

resulting in *exmpl.lib* having the order

```
obj3    obj5    obj1    obj2    obj4    obj6
```

And the following command will move *obj2* to the end of the library:

```
lb exmpl -ma+ obj2
```

3.3 Deleting Modules

Modules can be deleted from a library using *lb*'s *-d* option. 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    obj6
```

The following command deletes *obj3* and *obj5* from this library:

```
lb exmpl -d obj3 obj5
```

3.4 Replacing Modules

The *lb* option 'replace' 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
```

3.5 Uniqueness

lb allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

The option *-u* causes *lb* to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, the *-u* option 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
```

3.6 Extracting modules from a Library

The *lb* 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 follows the *-x* option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it's 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.

3.7 The 'verbose' option

The 'verbose' option, *-v*, causes *lb* to be verbose; that is, to tell you what it's doing.

This option can be specified as part of another option, or all by itself. For example, the following command creates a library in a chatty manner:

```
lb exmpl -v mod1 mod2 mod3
```

And the following equivalent commands causes *lb* to remove some modules and to be verbose:

```
lb exmpl -dv mod1 mod2
lb exmpl -d -v mod1 mod2
```

3.8 The 'silence' option

The 'silence' option, *-s*, causes *lb* to not display its signon message.

This option is especially useful when redirecting the output of a list command to a disk file, as described below.

3.9 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.*

3.10 Help

The *-h* option is provided for brief lapses of memory, and will generate a summary of *lb* functions and options.

NAME

lock, unlock - lock and unlock files

SYNOPSIS

lock file1 file2 ...

unlock file1 file2 ...

DESCRIPTION

These commands lock and unlock the specified files. When locked, a file can't be removed, renamed, or written to.

The code for these commands is contained in the SHELL.

NAME

ls - list directory contents

SYNOPSIS

ls [-options] [name1 name2 ...]

DESCRIPTION

ls displays information about the files and directories *name1*, *name2*, ... If no names are specified, *ls* displays information about all the files and directories in the current directory. For example, the following command displays information about the files *sub1.o* and *sub1.c* in the current directory, and the files in the directory */p1/include*:

```
ls sub1.o sub1.c /p1/include
```

A name can optionally specify multiple files, using the "wildcard characters" * and ?. These have their standard meaning: * matches one or more characters, and ? matches a single character. For example, the following command displayes information about all files that have extension *.c* and that are in the directory */test/src*:

```
ls /test/src/*.c
```

ls sends the information to its standard output. This information thus by default is sent to the console, but can be redirected to a file or other device in the normal way. For example, the first of the following commands displays on the console information about files that have extension *.o* and that are in the current directory. The second command sends information about the same files to the file *info.obj*:

```
ls *.o
ls *.o >info.obj
```

ls by default displays information in 'short form', listing just the names of the specified files and directories. You can also specify the *-l* option to cause *ls* to display information in 'long form', listing lots of information.

When *ls* sends information in short form to the console, the names are in columns on the screen, with a dash preceding directory names. When the information is sent to a file or other device, the names are listed one per line, and a directory name isn't by default preceded by a dash.

ls usually sorts the list it's going to display. By default, the list is sorted alphabetically; you can also specify options to cause *ls* to sort based on other the list such as 'last modified' time and file size, and, for a given criteria, to sort in the reverse of the normal order.

ls supports the following options:

-l	List in long form. For a description of the 'long form' information, see below.
-p	When listing in short form, precede directory names with a dash.
-t	Sort by 'last-modified' time.
-s	Sort by file size.
-r	Reverse the order of the sort. For example, when sorting alphabetically, list names beginning with 'z' first and those beginning with 'a' last.
-x	Don't sort the file list.

1. Long format

The *-l* option causes the listing to be made in 'long format', in which additional information is displayed for each file. In this case, the listing for a file or directory has the following format:

```
flags  type  (aux_type)  size  date  name
```

where:

* *name* is the name of the file or directory;
* *date* is the date and time at which it was last modified;
* *size* is the number of bytes that have been written to it;
* *aux_type* is its aux_type field (for a program, this is its load address);
* *type* defines the contents of the file or directory;
* *flags* defines other attributes of the file or directory.

The following paragraphs define the meanings of the *flags* and *type* fields.

1.1 The *Flags* Field

The *Flags* field consists of five characters, each of which defines whether or not the file has a certain attribute. If the file or directory has all these attributes, the Flags field will be "debwr". If the file or directory doesn't have a particular attribute, the attribute's character is replaced in this list by a dash, '-'.

The meanings of the characters are:

d	Deletable.
e	Erasable.
b	Needs to be backed up.
w	Writable.
r	Readable.

1.2 The *Type* Field

The type field defines the type of the file. Possible values:

PRG	File contains a program that can only be run in the SHELL environment.
BIN	File contains a type program that can be run in the SHELL or Basic environment.
SYS	File contains a ProDOS system file.
DIR	Directory.
TXT	This is the only other type of file created by the Aztec programs. *fopen*, *open*, and related functions also create files of type TXT.

2. Mounted volumes

The SHELL pretends that the file system has a root directory, and that all volume directories are subdirectories of this root directory. So to display the names of all mounted volumes, enter the command

```
ls /
```

And to display the names of all mounted volumes along with the numbers of the slots and drives that contain them, enter the command

```
ls -l /
```

NAME

mkdir - make directory

SYNOPSIS

mkdir dirname1 dirname2 ...

DESCRIPTION

mkdir creates one or more directories, named *dirname1*, *dirname2*, ...

SEE ALSO

An empty directory can be removed with the *rm* command.

NAME

mv - move files

SYNOPSIS

mv [-f] infile outfile

mv [-f] file1 [file2 ...] dir

DESCRIPTION

mv moves files, and their attributes. The original files then cease to exist.

The first form of the command, as shown above, copies *infile* to *outfile*. The second copies *file1, file2,* ... into the directory named *dir.*

The *-f* option causes *mv* to automatically overwrite any existing files. If this option isn't specified and if a file to be created already exists, *mv* will ask if you want it overwritten.

In both its forms, *mv* will simply change the name of the original file to that of the target file if the two files are in the same directory. It physically copies a file and then deletes the original only when the directories of the two files differ.

For example, the following command moves the file *hello.c* that is in the current directory to the file *newfile.c* in the */source* directory.

```
mv hello.c /source/newfile.c
```

The next command moves all ".lib" files in the */ln* directory to the */p1/lib* directory:

```
mv /ln/*.lib /p1/lib
```

NAME

obd - list object code

SYNOPSIS

obd <objfile>

DESCRIPTION

obd lists the loader items in an object file. It has a single parameter, which is the name of the object file.

NAME

ord - sort object module list

SYNOPSIS

ord [-v] [infile [outfile]]

DESCRIPTION

ord sorts a list of object file names. A library of the object modules that is generated from the sorted list by the object module librarian, *lb*, will have a minimum number of 'backward references'; that is, global symbols that are defined in one module and referenced in a later module.

Since the specification of a library to the linker causes it to search the library just once, a library having no backward references need be specified just once when linking a program, and a library having backward references may need to be specified multiple times.

infile is the name of a file containing an unordered list of file names. These files contain the object modules that are to be put into a library. If *infile* isn't specified, this list is read from *ord*'s standard input. The file names can be separated by space, tab, or newline characters.

outfile is the name of the file to which the sorted list is written. If it's not specified, the list is written to *ord*'s standard output. *outfile* can only be specified if *infile* is also specified.

The *-v* option causes *ord* to be verbose, sending messages to its standard error device as it proceeeds.

NAME

pr - initialize devices

SYNOPSIS

pr s1 [s2 ...]

DESCRIPTION

pr initializes the devices that are in slots *s1*, *s2*, ..., by calling each device's ROM.

For example, the following command initializes the card in slot 2 by calling address 0xc200:

```
pr 2
```

NAME

pwd - "print working directory"

SYNOPSIS

pwd

DESCRIPTION

pwd prints the name of the current directory.

The name is written to pwd's standard output device. Hence, the name is printed on the screen, by default, and can be redirected to another device or file, if desired.

The code for *pwd* is contained in the SHELL.

SEE ALSO

cd

NAME

rm - remove files and directories

SYNOPSIS

rm file [file] ...

DESCRIPTION

rm removes the specified files and directories.

rm will not remove locked files or non-empty directories.

For example, the following command removes the files *file1.bak* and *file2.bak* from the current directory:

```
rm file1.bak file2.bak
```

NAME

set - environment variable and exec file utility

SYNOPSIS

set

set VAR=string

set [-+x] [-+e] [-+n]

DESCRIPTION

set is used to examine and set environment variables, to set exec file options, and to enable the trapping of errors by the SHELL.

set is a builtin command; that is, its code is contained in the SHELL.

Displaying and setting environment variables

The first form listed for *set* causes *set* to display the name and value of each environment variable.

The second form assigns *string* to the environment variable *VAR*.

Setting Exec file options

The third form, which can only be used within an exec file, sets options for the exec file. The options are associated with a character, as follows:

x	Command line logging. With this option enabled, before a command line in an exec file is executed, it's logged to the screen. By default, this option is disabled.
e	Exit prematurely. With this option enabled, a command which terminates with a non-zero return code causes the exec file to be aborted. By default, this option is enabled.
n	Non-execution. With this option enabled, commands in the exec file aren't executed. By default, this option is disabled.

Preceding an option's character with a minus sign enables the option, and preceding it with a plus sign disables it.

NAME

shift - shift exec file variables

SYNOPSIS

shift [n]

DESCRIPTION

shift causes the values assigned to an exec file variable to be reassigned to the next lower-numbered exec file variable. *n* is the number of the lowest-numbered variable whose value is to be reassigned, and defaults to 1.

Thus,

```
shift
```

causes the exec file variable $1 to be assigned the value of $2, $2 to be assigned the value of $3, and so on. The original value assigned to $1 is lost. When all arguments to the exec file have been shifted out, $1 is assigned the null string.

EXAMPLES

The following exec file, *del*, is passed a directory as its first argument and the names of files within the directory that are to be removed:

```
set j = $1
shift
loop i in $*
rm $j/$i
eloop
```

In this example, *j* is an environment variable. The first two statements in the exec file save the name of the directory and then shift the directory name out of the exec file variables.

The loop then repeatedly calls *rm* to remove one of the specified files from the directory.

Entering

```
del file1.bak file2.bak
```

will remove the files *file1.bak* and *file2.bak* from the current directory.

NAME

sqz - squeeze an object library

SYNOPSIS

sqz file [outfile]

DESCRIPTION

sqz compresses an object module that was created by the Manx assembler.

The first parameter is the name of the file containing the module to be compressed. The second parameter, which is optional, is the name of the file to which the compressed module will be written.

If the output file is specified, the original file isn't modified or erased.

If the output file isn't specified, *sqz* creates the compressed module in a file having a temporary name, erases the original file, and renames the output file to the name of the original file. The temporary name is derived from the input file name by changing it's extent to *.sqz*.

If the output file isn't specified and an error occurs during the creation of the compressed module the original file isn't erased or modified.

NAME

tty - terminal emulation program

SYNOPSIS

tty -syy -bxx

DESCRIPTION

tty is a terminal emulation program that allows an Apple // operator to talk to another computer. To the other system, the Apple // will appear to be a terminal that supports some of the special features of the ADM-3A terminal.

tty reads characters from the keyboard and writes them to a serial interface. It also reads characters from this interface and writes them to the console. This interface must be compatible with the Super Serial Card.

The *-s* option defines the number of the slot containing the interface. The number immediately follows the *-s*, with no intervening spaces. If this option isn't specified, the interface is assumed to be in slot 2.

The *-b* option defines the baud rate of the serial interface. The baud rate immediately follows the *-b* option, with no intervening spaces. If this option isn't specified, the baud rate is assumed to be 9600.

To exit *tty*, type control-2; ie, type the '2' key while holding down the control key.

NAME

ved - Vi-like text editor

SYNOPSIS

ved [-tn] [-gprog] [file] [+l,c msg]

DESCRIPTION

ved is a screen oriented text editor that has some of the features of the UNIX Vi editor.

This description of *ved* has two sections: the first is a tutorial, showing how *ved* can be used to create a simple program. The second section then describes the features of *ved* in detail.

1. VED Tutorial

In the following paragraphs, we will use *ved* to create a short C program. The following is a listing of the program:

```
main(argc, argv)
int argc;
char *argv[];
{
    register int i = 1;

    printf("Program <%s> has %d arguments\n", argv[0], argc-1);
    while (--argc) {
        printf("Arg %d = <%s>\n", i, argv[i]);
        i++;
    }
}
```

As can be seen, the program prints its name, which is the first argument, and the number of arguments. Since the number of arguments includes the program name, argc-1 is used as the number of real arguments. Then, each argument is listed on a separate line.

To start *ved* type:

```
ved args.c
```

ved will be loaded from the current execution drive, and will try to find *args.c* on the disk. When it doesn't find it, it will say so and will start with an empty document. Note that the screen should look like:

```
"args.c" line 1 of 1

-
-
-
```

The cursor should be on the second line, and a single '-' on all the remaining lines. The '-' indicates that the line is after the end of the file.

ved has two modes, command and insert. Normally, *ved* is in command mode. For a list of most of the commands available, try typing a question mark without a return. The screen should clear, and the list should appear. Pressing the return key should repaint the screen with the document being edited. To enter insert mode, simply press the 'i' key. On the status line, the <INSERT> mode indicator should appear. This will always be there when in insert mode.

At this point, type in the test program, using the left arrow key to correct any mistakes. The indentation in the program is produced by using a tab character. The tab width defaults to four. It can be changed using the *-t* option when the editor is started. For example, starting *ved* with the command

```
ved -t8 file.c
```

sets tab stops every eight characters.

In order to use VED on older Apple 2's, which don't have a full ASCII keyboard, you need to have the single wire switch key modification installed. You can then enter upper and lower case alphabetic characters using the SHIFT key. You can enter the special character used by C programs by typing a control character; that is, by holding down the control key and then typing another key. The following table lists these control characters and the characters to which they are translated. In this table, ^X is an abbreviation for "type X while holding the control key down". The first column identifies control codes that you type; the second identifies the characters to which control codes are translated when the keyboard is in lower case mode (that is, when the SHIFT key is off); and the last column identifies the characters to which control codes are translated when the keyboard is in upper case mode.

Press:	*To get (lower):*	*To get (upper):*
^P	`	@
^A	{	[
^E	|	\
^R	}	]
^N	~	^
^C	DEL	_
^Q	ESC	
^U	TAB	

^U is generated by typing the 'right arrow' key.

To exit insert mode, type is ESC key. In addition to the ESC key, you can type control-Q (ie, type Q while holding down the CTRL key) to exit VED's 'insert' mode. This is useful on some older Apple 2e's, for which ESC is intercepted by the ROM and never gets back to VED.

Once out of insert mode, the cursor can be moved around using the space bar to move right and the left arrow to move left. To move a number of characters to the right or left, type the number of characters to skip followed by the space or backspace. To move to the beginning of the next line, use the return key. Similarly, use the '-' key to move to the beginning of the previous line. Characters can be deleted by placing the cursor on the character and pressing the 'x' key. Characters can be inserted by placing the cursor at the insertion point and pressing 'i' to enter insert mode.

When the program has been entered and corrected, type *:w* followed by a return. This will write the document to the file we originally tried to edit, *args.c*. To write the document to a different file, simply type *:w file.c* followed by a return.

When the file has been written, exit the editor by typing *:q* followed by return. If you try to exit without writing the file, *ved* will display the message:

```
file modified - use q! to override
```

This message will appear whenever you try to exit *ved* after making a change without writing the file out. To exit without saving the changes made, type *:q!* followed by return.

2. VED Reference Section

If *ved* is invoked with a file name, that file will be loaded into the memory buffer, otherwise it will be empty. *ved* will only edit text files: binary files cannot be edited. *ved* does all its editing in memory and is thus limited in the size of files that it will edit. In *ved*, the memory buffer is never completely empty. There will always be at least one newline in the buffer.

2.1 The screen

ved has a 1000 character limit on the size of a line. If a line is longer than the width of the screen, it will wrap to the next line. If a line starts at the bottom of the screen, and is too wide to fit, the line will not be displayed. Instead, the '@' character will be displayed. Likewise, at the end of the file, all lines beyond the end will consist only of a single '-' on each line.

A number of commands take a numeric prefix. This prefix is echoed on the status line as it is typed.

2.2 Moving around in the file

The normal mode of *ved* is command mode. During command mode, there are a number of ways to move the cursor around the screen and around the whole file.

newline	-	move to the beginning of the next line.

-	-	move to the start of the previous line.
space	-	move to the next character of the line.
backspace	-	move to the previous character.
0	-	move to the first character of this line.
$	-	move to the last character of this line.
h	-	move to the top line of the screen.
l	-	move to the bottom line of the screen.
b	-	move to the first line of the file.
g	-	move to the n'th line of the file.
/string	-	move to the next occurrence of 'string'.

2.3 Deleting text

When the cursor is in the appropriate spot, there are two commands used to delete existing text.

x	-	delete characters on the current line, beginning at the cursor and continuing up to, but not including, the newline.
dd	-	delete lines starting with the current line.

The *x* and *dd* commands can be prefixed with a number, which defines the number of characters or lines to be deleted. If a number isn't specified, just one character or line is deleted. For example, the first of the following commands deletes one line and the second deletes 10 lines:

```
dd
10dd
```

Note that deleting the last character on the line (newline character) causes the following line to be appended to the current line.

2.4 Inserting text

To add new text, hitting the 'i' key will cause the top line of the screen to indicate that you are now in <INSERT> mode. To exit insert mode, type ESCAPE. To insert a control character which means something special to *ved* into a text file, first type control-v followed by the control character itself. Control characters are displayed as '^X', where X is the appropriate character.

Typing 'o' will cause a new line to be created below the current line, and the cursor will be placed on that line and the editor placed into <INSERT> mode.

2.5 Moving text around

There are three commands used for moving text around. These commands make use of a 1000 character yank buffer. The contents of this buffer is retained across files.

yy	-	yank lines starting with the current line into the yank buffer.

yd	-	yank lines starting with the current line and then delete them.
p	-	"put" the lines in the yank buffer after the current line. The yank buffer is not modified.

A number can be prefixed to the yank commands, defining the number of lines to be yanked. If a number isn't specified, just one line is yanked. For example, the first of the following commands yanks one line and the second yanks and then deletes 5 lines.

```
yy
5yd
```

2.6 Miscellaneous commands

The 'z' command redraws the screen with the current line in the center of the screen.

The 'r' command replaces the character under the cursor with the next character typed.

2.7 File-related commands

When in command mode, if the ':' key is hit, a ':' will be displayed on the status line. At this point, a number of special file-related commands may be given.

:f	-	displays info about the current file.
:w file	-	writes the buffer to the specified file name.
:w	-	writes the buffer to the last specified file.
:e[!] file	-	clears the buffer and prepares *file* for editing.
:r file	-	reads the named file into the buffer.
:q[!]	-	exits the editor.

In the above table, square brackets surrounding a character indicate that the character is optional. The exclamation mark tells *ved* to execute the command in which it's specified, even if the file that's currently being edited has been modified since it was last written to disk.

LIBRARY FUNCTIONS OVERVIEW: APPLE // INFORMATION

Library Functions Overview: Apple // Information

The *Library Functions Overview* chapter presented overview information that is independent of the system on which your programs run. This chapter presents overview information about the library functions that is specific to programs that run on Apple //.

The sections of this chapter are numbered; the information discussed in a section is related to the section in the *Library Functions Overview* chapter that has the same number.

1. Overview of I/O: Apple // Information

A program running on Apple ProDOS can have at most eight files and devices open at once; this includes the standard i/o devices, and files and devices opened 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.

1.1 Pre-opened devices and command line arguments

Redirection of a program's standard i/o devices and the passing of arguments to it are only available to PRG programs that is, programs that only run under ProDOS in the SHELL environment. These features are not available to BIN and SYS ProDOS programs or to DOS 3.3 programs.

For a PRG-type program, the program's name is pointed at by the first item in the array that is pointed at by the second argument of the of the program's *main* function. That is, if the *main* function begins

```
main(argc, argv)
int argc; char *argv[];
```

then *argv[0]* is a pointer to the program's name.

For PRG-type programs that are activated by the SHELL, a command line argument can be a quoted string.

A PRG-type program can activate another program, by calling one of the *exec* functions that are described in the ProDOS functions chapter. If the called program is of type PRG, the calling program can pass arguments that the called program will see in its *main* function's *argv* array. Also, the called program will 'inherit' the files and devices that were left open for unbuffered i/o by the caller; that is, these files and devices are open when the called program is started, and it can access them using the same file descriptors as did the caller. For more details, see the *Command Programs* section in the *Technical Information* chapter.

1.2 File I/O

1.2.2 Random I/O

1.2.2.1 ProDOS Information

ProDOS keeps track of the last byte that has been written to a file. Because of this, the appending of data to a file and positioning of a file relative to its end by a program is always correctly done.

1.2.2.2 DOS 3.3 Information

On DOS 3.3, a program cannot always append data to a file or position the file relative to its end. This is discussed in the following paragraphs.

UNIX keeps track of the last character written to a file. Since the Aztec I/O functions attempt to make a file look like a UNIX file to a program, when a program requests that a file be positioned relative to its end (that is, relative to the last character which was written to it), the Aztec C routines must try to locate the last character which was written to it. This can always be done on ProDOS, since this this system keeps track of the last character written to a file.

However, DOS 3.3 only keeps track of the last record written to a file, and not the last character. Because of this, it is not always possible for the Aztec C i/o functions to determine the last character written to the file, when the program in which they are contained is running on DOS 3.3. And because of this, it is not always possible for a program running on DOS 3.3 to correctly position a file relative to its end.

When a program running on DOS 3.3 requests positioning of a file relative to its end, the Aztec i/o functions try to find the last character written to the file. They always succeed if the file contains only text; for files containing arbitrary data, they may not succeed.

To locate the last valid character in a file on DOS 3.3, the Aztec routines use the following fact: when a file is created on these systems using Aztec C, the last record in the file is padded at the end with the special character which denotes the end of a text file. For DOS 3.3, the special character is 0. If the program exactly filled the last record, it won't have any padding.

When a program requests that a file be positioned relative to its end, the Aztec C i/o routines search the file's last record; end of file is declared to be located at the position following the last non-end-of-file character.

For files of text, this algorithm always correctly determines the last character in the file, so appending to text files is always correctly done.

For other files, this algorithm will still correctly determine the last valid character in the file...most of the time. However, if the last valid characters in the file are end-of-file characters, the file will be incorrectly positioned.

1.2.3 Opening Files

To open a file on ProDOS, a program can use either a fully- or partially-qualified file name, as described in the chapter on the SHELL.

When a file is created on ProDOS, its type is set to TXT, and flags are set allowing it to be read, written, renamed, and erased. To change these attributes, a program can fetch and reset them, using the *getfinfo* and *setfinfo* functions.

There are two descriptions of the *open* and *creat* functions in the manual. You should read the descriptions that are in the Apple // Functions chapter, since they present the special Apple // features of these functions. The descriptions in the System Independent Functions chapter aren't complete, since they don't discuss these special features.

1.3 Device I/O

Two groups of devices are supported by Aztec C for Apple //: devices that are of a particular type, and devices that are identified only by the slot in which they are located.

The devices of a particular type are:

device	*description*
con:	Console
pr:	Printer
ser:	Serial Device

The name of a "slot device" has the form *sn:*, where *n* is the number of the slot. Thus, *s2:* is the name of the device in slot two.

s0: is another name for the console. There are thus two ways to access the console: as the *con:* device; and as the *s0:* device. We'll discuss these two ways below.

A program can issue a request to access a certain device without knowing the specific attributes of the device. The Aztec routine that services the request, (called a "device driver") will translate the request into a sequence of device-dependent operations. For this, it uses information about the devices on your system that you provided using the *config* program or that it itself detected. For information, see the Devices section in the SHELL chapter and the description of *config* in the Utility Programs chapter.

1.3.1 The Console

As mentioned above, a program can access the console using either the *con:* or the *s0:* device. *con:* provides a program "high level" access to the console, allowing a program to easily perform console i/o in a variety of ways. The features of console i/o that are described in the System Independent Library Overview chapter are available to programs that access the console using *con:*. Those features are system independent: a program using these features can run on any system supported by Aztec C. Other features of console i/o when using the *con:* device, which may or may not be available on other systems supported by Aztec C, are described in the *Console I/O* section of this chapter.

s0: provides a program "low level" access to the console. An i/o request to *s0:* is simply passed on to the Apple's console i/o routines for processing (*cout* for output, *keyin* for input), with the Aztec driver performing some manipulation on the i/o data, if the console requires it (as defined by you using the *config* program). These manipulations are the same that are available when i/o is performed to any slot device; for information, see the discussion of slot devices, below.

1.3.2 I/O to Other Devices

1.3.2.1 The Printer

pr:, the printer device, is associated with a slot device and an initialization string (as defined by you using the *config* program).

When a program opens *pr:*, the device that's in the associated slot will be initialized, just as if the slot device was being opened. Then the initialization string is sent to the device.

Output requests to *pr:* are performed just as if the request was sent to the device's slot device.

1.3.2.2 The Serial Device

The serial device, *ser:*, is associated with a slot device (as defined by you using the *config* program). I/O requests to *ser:* are processed just as if the request was made to the associated slot device.

1.3.2.3 The Slot Devices

There is a slot device for each slot in an Apple //: slot device *s1:* is associated with the device in slot 1, *s2:* with the device in slot 2, and so on. *s0:* is special, being associated with the console.

As mentioned above, the slot devices provide "low level" access to the devices in the slots: an i/o request to a slot device from a program is simply passed onto the device's ROM routine for processing, with some manipulations on the transferred data being performed by the Aztec slot driver if required (as defined by you using *config*).

The Aztec slot driver can distinguish between three types of slot devices: those using the Pascal 1.1 protocol, those using Pascal 1.0 protocol, and those using Basic protocol. When a program opens the device, the Aztec slot driver determines the I/O protocol used by the device and calls the appropriate ROM routine to initialize the device.

4. Overview of Console I/O using *con:* Apple // Information

This section discusses features of the *con:* device that aren't discussed in the Library Overview chapter.

On Apple //, the *con:* device supports the UNIX console i/o options that are described in the Library Overview chapter. In addition, other options are supported by *con:* that aren't UNIX-compatible, including the automatic expansion of tabs to spaces on output (the XTAB option) and whether or not the program will wait if it issues a read to the console when a key hasn't yet been depressed (the NODELAY option).

A program's default console mode on Apple // is line-oriented, with ECHO and CRMOD enabled, just as it would be on another system. In addition, an Apple // program's default mode has the special options XTAB and ECHOE (defined below) enabled, NODELAY disabled, and tab stops set every four characters.

4.1 Line-oriented Input

On Apple //, all console options are program-selectable, even in line-oriented input mode.

Thus, line-oriented input doesn't automatically enable ECHO for a Apple // program.

On Apple //, a non-UNIX option, NODELAY is available, which defines whether a program wants to wait if its read request can't be immediately satisfied. With NODELAY reset and with the console in line-oriented mode, a read request to the console will wait if an entire line hasn't been typed. With NODELAY set and with the console in line-oriented mode, a read request will always return immediately: if an entire line hasn't been typed, no characters will be returned to the program (even if some characters have been typed); if an entire line has been typed, the requested characters will be returned.

4.2 Character-oriented Input on Apple //

On Apple //, there is one exception to the rule that RAW mode resets all other options: with the console in RAW mode, a program still has control over the NODELAY option.

With the console in character-oriented input mode, the driver's treatment of a read request to the console depends on the console's NODELAY option: if this option is reset, the program will be suspended until at least one character has been received; then, the requested number of characters, up to the number in the internal buffer, are returned to the program. Thus, suppose a program issues the input call

```
        read (0, buf, 80)
```

to the console, which is in a character-oriented mode with NODELAY reset. If there are characters in the driver's internal buffer, it will return the requested number of characters from this buffer, up to the number in the buffer; if 80 characters aren't already in the buffer, it won't wait for the operator to enter the remaining characters. If there are no characters in the driver's buffer, the driver will suspend the program until the operator types a character, and then return that character to the program.

If the console is in character-oriented mode with NODELAY set, a read request to the console will always return immediately: if no characters are in the driver's buffer, no characters are returned to the program; otherwise, the characters in the buffer are returned, up to the number requested.

4.4 The *sgtty* fields

4.4.1 The *sg_flags* field

On Apple //, the following non-UNIX flags for *sg_flags* are supported in addition to the UNIX-compatible flags:

XTAB	Convert tabs to spaces on output, with tab stops set as specified by TABSIZ. By default, XTAB is enabled.
TABSIZ	A mask for a four-bit field that defines the tabwidth to be used when XTAB is set. By default, TABSIZ is set to four.
ECHOE	When ECHO is set, and the 'erase' character is entered, output the 'erase' character, then a space, and then another 'erase' character (thus erasing the character from the screen); By default, ECHOE is enabled.
NODELAY	When a read is issued to the console and no keys have been typed, return immediately. By default, NODELAY is disabled.

4.4.2 The *sg_erase* field

This field is supported on Apple //. When a program reads from *con:* and the console is in line-oriented mode, receipt of the *sg_erase* character causes the console driver to "erase" the last-typed character, by backspacing the cursor over it and by removing the last-typed character from its internal buffer.

By default, this character is ^H (that is control-H), or equivalently, the left-arrow key.

4.4.3 The *sg_kill* field

sg_kill is supported on Apple //. When a program reads from *con:* and the console is in line-oriented mode, receipt of the *sg_kill* character causes the console driver to "erase" the line that's currently being entered, by moving the cursor to the beginning of the next line and by deleting all characters in its internal buffer that haven't yet been returned to the program.

By default, this character is ^X, that is, control-X.

4.6 Screen Control Codes

Most characters that a command program sends to *con:* are simply written to the screen. Some, however, are control codes that cause the console driver to perform special functions. The control codes (in hex) and their functions are:

code	*function*
07	beep
08	non-destructive backspace
09	tab character
0a	cursor down/linefeed (scroll at bottom)
0b	cursor up
0c	non-destructive cursor right
0d	return to beginning of line
1a	home and clear screen
1e	home the cursor
1b 45	insert blank line at cursor
1b 51	insert blank character at cursor
1b 52	delete line at cursor
1b 54	clear to end of line from cursor
1b 57	delete character at cursor
1b 59	clear to end of screen
1b 3d y+20 x+20	move cursor to x,y position

Note: the "insert character" and "delete character" operations are not availble when you're using the standard Apple // console.

6. Error codes

The following table lists the ProDOS-specific error codes that may be returned to a program.

hex code	*Meaning*
00	No error
01	Invalid number for system call
04	Invalid param count for system call
25	Interrupt vector table full
27	I/O Error
28	No device connected/detected
2b	Disk write protected
2e	Disk switched
40	Invalid characters in pathname
42	File control block table full
43	Invalid reference number
44	Directory not found
45	Volume not found
46	File not found
47	Duplicate file name
48	Volume Full
49	Volume directory full
4a	Incompatible file format
4b	Unsupported storage type
4c	End of file encountered
4d	Position out of range
4e	File Access error; eg, file locked
50	File is open
51	Directory structure damaged
52	Not a ProDOS disk
53	Invalid system call parameter
55	Volume control block table full
56	Bad buffer address
57	Duplicate volume
5a	Invalid address in bit map

Apple // FUNCTIONS

Chapter Contents

Apple // Functions

This chapter describes functions which are available only to programs which are running on an Apple //, running either ProDOS or DOS 3.3.

The header to the description of a set of functions defines the environments in which the functions can be used, as does the index that follows this introduction.

This chapter is divided into sections, each of which describes a group of related functions.

As with description of the system independent functions, the header to a section of this chapter has a parenthesised letter that specifies the library containing the section's functions. The codes and their related libraries are:

C	c.lib;
S	s.lib;
G	g.lib.

The Graphics Functions

The graphics functions described in this section can be used to draw points, lines, and ovals in the hi-res primary graphics page.

With the screen in full-screen hi-res mode, the screen is a matrix 280 dots wide by 192 dots high, with the top left dot having coordinates (0,0).

With the screen in mixed-mode, the bottom four lines of the screen are used to display text and the remainder of the screen is in hi-res mode. The hi-res part of the screen in this case is 280 dots wide by 160 high.

Index to Apple // Functions

This section lists the Apple-specific functions that are provided with Aztec C65. The list is sorted alphabetically by function name. For each function it gives the function's name, the title of the section in which the function is described, and a phrase describing the function's purpose.

function	*page*	*description*
access	ACCESS	determine file accessibility
assert	ASSERT	verify program assertion
black	COLOR	paint screen black
blue	COLOR	paint screen blue
brk	BREAK	set heap pointer
chdir	DIRECTORY	change current directory (ProDOS only)
circle	CIRCLE	draw circle
creat	CREAT	creat file on Apple //
ctop	CTOP	C-to-Pascal string conversion
drw, etc	LINE	draw line
execl, etc	EXEC	jump to program (ProDOS SHELL)
exit	EXIT	terminate program
_exit	EXIT	terminate program
fixnam	FIXNAM	convert file name (ProDOS only)
fscreen	MODE	select full-screen mode
getenv	GETENV	get env var (ProDOS SHELL only)
geteof	EOF	get file's EOF (ProDOS only)
getfinfo	FILEINFO	get file information (ProDOS only)
getprefix	PREFIX	get default prefix (ProDOS only)
green	COLOR	paint screen green
hgr	MODE	select hi-res mode
lineto, etc	LINE	draw line
mktemp	MKTEMP	make name for temporary file
mkdir	DIRECTORY	make a directory (ProDOS only)
mscreen	MODE	select mixed mode
open	OPEN	open file on Apple //
page1, page2	PAGE	Page selection
perror, etc	PERROR	write error message
plot, etc	POINT	plot point
plotchar	PLOTCHAR	plot character
ptoc	CTOP	Pascal-to-C string conversion
_system	SYSTEM	ProDOS system call (ProDOS only)
sbrk	BREAK	set heap pointer
scr_curs, etc	SCREEN	screen manipulation

NAME

access - determine accessibility of a file or directory

SYNOPSIS

```
int access (filename, mode)
char *filename;
int mode;
```

DESCRIPTION

access determines whether a file or directory can be accessed in the way that the calling function wants to access it. It can also be used to just test for the existence of a file or directory.

filename points to the name of the file or directory; this name optionally contains the drive and path of directories that must be passed through to get to the file or directory. If the drive component isn't specified, the file or directory is assumed to reside on the default drive. If the path component isn't specified, the file or directory is assumed to reside in the current directory on the specified drive.

mode is an *int* that specifies the type of access desired:

mode	*meaning*
4	read
2	write
1	execute (if a file) or search (if a directory)
0	check existence of the file or directory.

If the existence of the file or directory is being checked (ie, mode=0), *access* returns 0 if the file exists and -1 if it doesn't. In the latter case, *access* also sets the symbolic value ENOENT in the global integer *errno.*

When *access* is called to determine if a file can be accessed in a certain way (ie, *mode* isn't 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 *errno* that defines why the access is not permitted.

When asked, *access* says that a directory can be read or written; this means that a program can create and delete files on the directory, not that it can directly read and write the directory itself.

The symbolic values that *access* may set in *errno* when it's called with a non-zero *mode* parameter are:

errno	*meaning*
ENOTDIR	A component of the path prefix is not a directory.

ENOENT	The file or directory doesn't exist.
EACCES	The file or directory can't be accessed in the desired manner.

SEE ALSO

The "Errors" section of the Library Overview chapter discusses *errno.*

NAME

assert - verify program assertion

SYNOPSIS

#include <assert.h>

assert (expr)
int expr;

DESCRIPTION

assert is useful for putting diagnostic messages in a program. When executed, it will determine whether the expression *expr* is true or false. If false, it prints the message

Assertion failed: *expr*, file *fff*, line *lnnn*

where *fff* is the name of the source file and *nnn* is the line number of the *assert* statement.

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>*.

NAME

sbrk, brk, rsvstk - heap management functions

SYNOPSIS

```
void *sbrk(size)

brk(ptr)
void *ptr;

rsvstk(size)
```

DESCRIPTION

sbrk and *brk* provide an elementary means of allocating and deallocating space from the heap. More sophisticated buffer management schemes can be built using these functions; for example, the standard functions *malloc*, *free*, etc call *sbrk* to get heap space, which they then manage for the calling functions.

sbrk increments a pointer, called the 'heap pointer', by *size* bytes, and, if successful, returns the value that the pointer had on entry. Initially, the heap pointer points to the base of the heap. *size* is a signed *int*; if it is negative, the heap pointer is decremented by the specified amount and the value that it had on entry is returned. Thus, you must be careful when calling *sbrk*: if you try to pass it a value greater than 32K, *sbrk* will interpret it as a negative number, and decrement the heap pointer instead of incrementing it.

brk sets the heap pointer to *ptr*, and returns 0 if successful.

rsvstk sets the heap-stack boundary *size* bytes below the current top of stack, thus changing the amount of space allocated to the stack and heap.

SEE ALSO

The functions *malloc*, *free*, etc, implement a dynamic buffer-allocation scheme using the *sbrk* function. See the Dynamic Buffer Allocation section of the Library Functions Overviews chapter for more information.

The standard i/o functions usually call *malloc* and *free* to allocate and release buffers for use by i/o streams. This is discussed in the Standard I/O section of the Library Functions Overviews.

Your program can safely mix calls to the *malloc* functions, standard i/o calls, and calls to *sbrk* and *brk*, as long as the your calls to *sbrk* and *brk* don't decrement the heap pointer. Mixing *sbrk* and *brk* calls that decrement the heap pointer with calls to the *malloc* functions and/or the standard i/o functions is dangerous and probably shouldn't be done by normal programs.

For more information on the heap and its relationship to the other areas of a program, see the Memory Organization section of the Technical Information chapter.

ERRORS

If an *sbrk* or *brk* request would make the heap space pointer go past the end of the heap, the function will return -1 as its value, without modifying the heap space pointer.

NAME

circle, set_asp - circle-drawing functions

SYNOPSIS

circle (x, y, rad)
int x, y, rad;
set_asp (xasp, yasp)
int xasp, yasp;

DESCRIPTION

circle draws an oval on the primary hi-res graphics page, with center at *(x,y)*. By default, the horizontal and vertical radii of the oval are both *rad*, resulting in *circle* drawing a circle.

set_asp controls the eccentricity (ie, the "ovalness") of the figure drawn by *circle*: *circle* draws an oval whose horizontal radius is *rad* * *xasp* and whose vertical radius is *rad* * *yasp*, where *xasp* and *yasp* have the values defined by the last call to *set_asp*. If *set_asp* isn't called, *circle* will use the value 1 for *xasp* and *yasp*, resulting in in a circle of radius *rad* being drawn.

SEE ALSO

color, line, mode, page, plotchar, point

NAME

black, blue, green, violet - color selection functions

SYNOPSIS

black()
blue()
green()
violet()

DESCRIPTION

Each of these functions sets the screen in Hi-res, full-screen graphics mode, using the primary graphics page, and clears the screen.

The entire screen will be a single color, as determined by the function that is called. For example, calling *green* makes the entire screen green.

SEE ALSO

circle, line, mode, page, plotchar, point

NAME

creat - create a new file

SYNOPSIS

```
creat(name, pmode)
char *name;
int pmode;
```

DESCRIPTION

There are two descriptions of the *creat* function in this manual: the one in the System Independent Functions chapter (which is supplied with all versions of Aztec C) describes *creat*'s implementation on most systems, while the one you're now reading describes its implementation on the Apple //. So to learn about the Apple // version of *creat*, you can read just this description and ignore the one in the System Independent Functions chapter.

creat creates a file and opens it for unbuffered, write-only access. If the file already exists, it is truncated so that nothing is in it (this is done by erasing and then creating the file).

creat returns as its value an integer called a "file descriptor". Whenever a call is made to one of the unbuffered i/o functions to access the file, its file descriptor must be included in the function's parameters.

name is a pointer to a character string which is the name of the device or file to be opened. See the I/O overview section for details.
.P For ProDOS programs, the *pmode* parameter of the *creat* function is the file's access mode. The meanings of the bits in this parameter (bit 0 is the least significant bit, bit 15 is the most significant):

Bit number	*Meaning*
0	File can be read
1	File can be written
2-4	Reserved
5	File modified since last backup
6	File can be renamed
7	File can be deleted
8-15	Unused

Thus, to create a file on which all types of operations are permitted, set its mode parameter to 0xc3.

For DOS 3.3 programs, *pmode* is the file's type. Type codes:

param3, in hex	*File type*
00	Text
01	Integer basic
02	Applesoft basic
04	Binary
08	Relocatable
10	S-type file
20	A-type file
40	B-type file

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *creat* fails, it returns -1 as its value and sets a code in the global integer *errno.*

NAME

ctop, ptoc - C <-> Pascal string functions

SYNOPSIS

```
char *
ctop(str)
char *str;

char *
ptoc(str)
char *str;
```

DESCRIPTION

ProDOS expects character strings to be in Pascal format, in which a string consists of a leading byte containing the number of characters in the string, followed by the characters in the string. In C, on the other hand, a character string consists of the characters followed by a null character.

ctop and *ptoc* convert a string from C form to Pascal form and from Pascal form to C form, respectively. The converted string overlays the original string, and the function returns a pointer to the converted string.

NAME

geteof, seteof - get and set end-of-file position

SYNOPSIS

```
long geteof(fd)          /* ProDOS functions */
int fd;

long seteof(fd, pos)
int fd; long pos;
```

DESCRIPTION

geteof returns as its value the end-of-file value of the file whose file descriptor is *fd*, by issuing the GET_EOF ProDOS MLI function.

seteof sets the eof-of-file value for the file whose file descriptor is *fd* to *pos*, and returns *pos* as its value. To do this, *set_eof* issues the SET_EOF ProDOS MLI function.

NAME

execl, execv, execlp, execvp - program activation functions

SYNOPSIS

```
execl(name, arg0, arg1, arg2, ..., argn, 0)
char *name, *arg0, *arg1, *arg2, ...;

execv(name, argv)
char *name, *argv[];

execlp(name, arg0, arg1, arg2, ..., argn, 0)
char *name, *arg0, *arg1, *arg2, ...;

execvp(name, argv)
char *name, *argv[];
```

DESCRIPTION

These functions load, and transfer control to, another program. The called program is loaded on top of the calling program; thus, if the exec function succeeds, it doesn't return to the caller.

The functions can be called by PRG programs; that is, by programs that can only be run in the SHELL environment. The functions can start any type of program, including those that have not been created using the Aztec software.

The functions can also be called by BIN programs, when the programs are running in the SHELL environment. However, unlike BIN programs, these functions can't be used outside the SHELL environment; thus, you should be wary about using these functions in a BIN program.

The following paragraphs will first describe the parameters to the exec functions, then describe the differences between the functions, and finally discuss other features of the functions.

Parameters

name is the name of the file containing the program to be loaded. It can optionally specify, using the standard ProDOS syntax, the complete or a partial path of directories that must be passed through to get to the file.

The exec functions can pass arguments to the called program. *execl* and *execlp* build 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 *arg0* as *argv[0]*, *arg1* as *argv[1]*, and so on. By convention, *arg0* is the name of the program being called.

execv and *execvp* build a command line by concatenating the strings pointed at by *argv[0]*, *argv[1]*, and so on. The *argv* array must be 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]*. By convention, *argv[0]* is the name of the program being called.

The Functions

execl and *execv* load a program from the specified file: *execl* is useful when a fixed number of arguments are being passed to a program. *execv* is useful for programs which are passed a variable number of arguments.

execlp and *execvp* search a list of directories for the program to be loaded, beginning with the current directory. If the program isn't there, the directories specified in the PATH environment variable are searched.

Passing Open Files and Devices

When both the calling and the called programs are of type PRG, the following comments describe the passing of open files and devices between the programs:

* Files that are left open for unbuffered i/o in the calling program will be open for unbuffered i/o in the called program, and will have the same file descriptors.
* Except for files that are associated with the stdin, stdout, and stderr standard i/o devices, files left open 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 PRG program activates another using an exec 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 called program accesses these logical devices using standard i/o calls.

When both programs are not of type PRG, open files and devices can not be passed between the programs.

DIAGNOSTICS

If an exec function fails, for example because the file doesn't exist, it will return -1 as its value.

NAME

exit, _exit

SYNOPSIS

exit(code)

_exit(code)

DESCRIPTION

exit and *_exit* terminate the execution of a program and restart the operating system-type program that activated the program (that is, the SHELL or the Basic Interpreter). *exit* closes files opened for both standard and unbuffered i/o, calling *fclose* to close each file opened for standard i/o, and then calling *close* to close any other files that are open for unbuffered i/o. *_exit* closes files opened for unbuffered i/o, calling *close* to close each such file.

For a PRG program, *code* is its return code. The return code is set to 0 for a PRG program that terminates by either explicitly or implicitly returning from its *main* function. An exec file that starts a PRG program can test the program's return code and act accordingly.

The return code of a BIN program is always 0, regardless of the value specified in the call to *exit* or *_exit.*

NAME

getfinfo & setfinfo - get & set file information

SYNOPSIS

```
#include <prodos.h>          /* ProDOS Functions */

getfinfo (file, fp)
char *file; struct finfo *fp;

setfinfo (file, fp)
char *file; struct finfo *fp;
```

DESCRIPTION

getfinfo fetches information from the directory about a file named *file*, by issuing the GET_FILE_INFO ProDOS MLI call, and returns the information in the structure pointed at by *fp*.

setfinfo sets information in the directory about the specified file, as defined by the structure pointed at by *fp*. To do this, it issues the SET_FILE_INFO ProDOS MLI call.

file can optionally specify, using the standard ProDOS syntax, the complete or partial sequence of directories that must be passed through to get to the directory that contains the file.

The structure that is pointed at by *fp* has the following format:

```
struct finfo {
        unsigned char  access;
        unsigned char  file_type;
        unsigned short aux_type;
        unsigned char  storage_type; /* getfinfo only */
        unsigned short blocks_used; /* getfinfo only */
        unsigned short mod_date;
        unsigned short mod_time;
        unsigned short create_date; /* getfinfo only */
        unsigned short create_time; /* getfinfo only */
};
```

For the definition of these fields, see the description of the GET_FILE_INFO and SET_FILE_INFO ProDOS MLI calls in the ProDOS Technical Reference Manual.

DIAGNOSTICS

If no error occurs, these functions return 0 as their value. If an error occurs, they set a code in the global int *errno* and return -1 as their value.

NAME

fixnam - convert file name to fully-qualified name

SYNOPSIS

```
fixnam(in_name, buf)     /* ProDOS only */
char *in_name, *buf;
```

DESCRIPTION

fixnam converts the file name pointed at by *in_name* into a fully-qualified name consisting of the file name itself prefixed by the directories that must be passed through to get to the file. *in_name* can use "." to refer to the current directory, and ".." to refer to a parent directory.

The converted name is placed in the buffer pointed at by *buf.*

For example, suppose that the current directory being */work/source/input.* The first call to *fixnam* that follows places */work/source/input/indvr.c* in *inbuf.* The second places */work/source/output/outdvr.c* in *outbuf.*

```
fixnam("indvr.c",inbuf)
fixnam("../output/outdvr.c",outbuf)
```

DIAGNOSTICS

fixnam returns 0 if successful. If the input file name contains so many ".." references that the resultant directory is above the root directory, *fixnam* sets EINVAL in the global integer *errno* and returns -1 as its value.

NAME

getenv - Get value of environment variable

SYNOPSIS

```
char *getenv(name)      /* ProDOS SHELL Function */
char *name;
```

DESCRIPTION

getenv, returns a pointer to the character string associated with the environment variable *name*, or 0 if the variable isn't in the environment.

The character string is in a dynamically-allocated buffer; this buffer will be released when the next call is made to *getenv*.

getenv can be called by PRG programs; that is, by programs that can only be run in the SHELL environment. It can also be called by BIN programs that are running in the SHELL environment. However, unlike BIN programs, *getenv* cannot be used outside of the SHELL environment; thus, you should be wary of calling *getenv* in a BIN program.

NAME

drw ..., lineto ... - line-drawing functions

SYNOPSIS

drw (x1, y1, x2, y2)
bdrw (x1, y1, x2, y2)
gdrw (x1, y1, x2, y2)
rdrw (x1, y1, x2, y2)
vdrw (x1, y1, x2, y2)

lineto (x, y)
blineto (x, y)
glineto (x, y)
rlineto (x, y)
vlineto (x, y)

DESCRIPTION

These functions draw straight lines on the primary hi-res graphics page. The "drw" functions (ie, the first five functions) draw a line from the point whose coordinates are *(x1, y1)* to *(x2, y2)*, differing in the color of the line, as follows:

function	*color*
drw	white
bdrw	blue
gdrw	green
rdrw	red
vdrw	violet

The "drw" functions set the global variables *_oldx* and *_oldy* to *x2* and *y2*, respectively.

The "lineto" functions (ie, the last five functions) draw a line from the point whose coordinates are (*_oldx*, *_oldy*) to the point *(x, y)*, and then set *_oldx* and *_oldy* to *x* and *y*, respectively. These functions differ in the color of the drawn line, as follows:

function	*color*
lineto	white
blineto	blue
glineto	green
rlineto	red
vlineto	violet

SEE ALSO

circle, color, mode, page, plotchar, point

NAME

mkdir - make directory

SYNOPSIS

```
mkdir (name)          /* ProDOS function */
char *name;
```

DESCRIPTION

mkdir creates the directory named *name*.

name can optionally specify, using the standard ProDOS syntax, the complete or partial sequence of directories that must be passed through to get to the directory that is to be the parent of the created directory.

DIAGNOSTICS

If no error occurs, *mkdir* returns 0 as its value. If an error occurs, it sets a code in the global int *errno* and returns -1 as its value.

NAME

mktemp - make a unique file name

SYNOPSIS

char *
mktemp (template)
char *template;

DESCRIPTION

mktemp replaces the character string pointed at by *template* with the name of a non-existent file, and returns as its value a pointer to the string.

The string pointed at by *template* should look like a file name whose last few characters are *X*s with an optional imbedded period.

mktemp replaces the *X*s with a letter followed by digits. The digits are set to the address of the program's _*main* function. The letter will be between 'A' and 'Z', and will be chosen such that the resulting character string isn't 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 *template* 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 file name. If the program's _*main* function begins at decimal address 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")==0)
   if (!*fname){
      printf("mktemp failed");
      exit(1);
   } else
      fp=fopen(fname, "w");
   ...
}
```

NAME

text, hgr, fscreen, mscreen - mode-selection functions

SYNOPSIS

text ()
hgr ()
fscreen ()
mscreen ()

DESCRIPTION

text sets the screen in text mode.

hgr sets the screen in Hi-res graphics mode. Unlike the Color functions, *hgr* doesn't clear the screen.

fscreen gives you a full screen to work with in the graphics mode. This means that you have a 280 by 192 matrix to work with.

mscreen sets the screen in mixed text and Hi-res modes. In this mode, the four lines at the bottom of the screen are used to display text, and the remainder of the screen (a 280 by 160 matrix) is in Hi-res mode.

SEE ALSO

circle, color, line, page, plotchar point

NAME

open

SYNOPSIS

#include "fcntl.h"

open(name, mode, param3) /* Apple // calling sequence */
char *name;

DESCRIPTION

There are two descriptions of the *open* function in this manual: the System Independent Functions chapter (which is supplied with all versions of Aztec C) describes the *open* function that is provided for most systems, while the one you're now reading describes the *open* function that is provided for the Apple //. So to learn about the Apple // version of *open*, you can read just this description and ignore the one in the System Independent Functions chapter.

open opens a device or file for unbuffered i/o. It returns an integer value called a file descriptor which is used to identify the file or device in subsequent calls to unbuffered i/o functions.

name is a pointer to a character string which is the name of the device or file to be opened. For details, see the overview section I/O.

mode specifies how the user's program intends to access the file. The choices are as follows:

mode	*meaning*
O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read and write
O_CREAT	Create file, then open it
O_TRUNC	Truncate file, then open it
O_EXCL	Cause open to fail if file already exists; used with O_CREAT
O_APPEND	Position file for appending data

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.

The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, and O_RDWR in the mode parameter. The three remaining values are optional. They may be included by adding them to the mode parameter, as in the examples below.

By default, the open will fail 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. For systems which don't keep track of the last character written to a file (for example, CP/M and Apple DOS), this positioning cannot always be correctly done. See the I/O overview section for details. Also, this option is not supported by UNIX.

When *open* is used to create a file under ProDOS, *param3* is the file's access mode. The meanings of the *param3* bits, where bit 0 is the least significant bit, bit 15 is the most significant:

Bit number	*Meaning*
0	File can be read
1	File can be written
2-4	Reserved
5	File has been modified since last backup
6	File can be renamed
7	File can be deleted
8-15	Unused

Thus, to create a file on which all types of operations are permitted, set its *param3* parameter to 0xc3.

When *open* is used to create a file under DOS 3.3, *param3* is the file's type:

param3, in hex	*File type*
00	Text
01	Integer basic
02	Applesoft basic
04	Binary
08	Relocatable
10	S-type file
20	A-type file
40	B-type 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.

SEE ALSO

I/O (O), Unbuffered I/O (O), Errors (O)

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, 0);
```

The third parameter of *open* is not important in this case, since *open* won't create the file. If *testfile* does not exist *open* will just return -1 and set *errno* to ENOENT.

2. To open the file *sub1* on ProDOS in read-write mode, allowing it to be deleted, read, written, and renamed:

```
fd = open("sub1", O_RDWR+O_CREAT, 0xc3);
```

If the file does not exist, it will be created and then opened.

3. The following program opens a ProDOS file whose name is given on the command line. The file must not already exist.

```
main(argc, argv)
char **argv;
{
    int fd;

    fd = open(*++argv,
            O_WRONLY+O_CREAT+O_EXCL, 0xc3);
    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");
}
```

NAME

page1, page2 - page-selection functions

SYNOPSIS

page1 ()
page2 ()

DESCRIPTION

page1 and *page2* enable the primary and secondary pages, respectively.

SEE ALSO

circle, color, line, mode, plotchar, point

NAME

perror, errno - system error messages

SYNOPSIS

```
int perror (s)
char *s;

#include <errno.h>

extern int errno;
```

DESCRIPTION

When a library function detects an error, it will generally set an error code, which is a positive integer, in the global integer *errno* and return an appropriate, function-dependent value.

The *extern* declaration of *errno* is in *errno.h.*

When an error occurs, *perror* can be called to write a message describing the error on the standard error device. The message consists of the following:

* *s*, the string pointed at by the argument to *perror*,
* a colon and a blank,
* the *sys_errlist* message corresponding to the current value of *errno*,
* a newline character.

perror returns 0 if *errno* contains a valid value; otherwise it returns -1 without printing a message.

SEE ALSO

Error Overview (O)

NAME

plotchar - plot character

SYNOPSIS

plotchar (c, x, y)
int c, x, y;

DESCRIPTION

plotchar displays the printable char *c* on the primary hi-res graphics page, at the location having coordinates *(x, y)*.

SEE ALSO

circle, color, line, mode, page, point

NAME

plot, ... - point-plotting functions

SYNOPSIS

plot (x, y)
bplot (x, y)
gplot (x, y)
rplot (x, y)
vplot (x, y)

DESCRIPTION

These functions plot a point on the primary hi-res page, at the location *(x, y)*. They differ in the color of the point:

function	*color*
plot	white
bplot	blue
gplot	green
rplot	red
vplot	violet

SEE ALSO

circle, color, line, mode, page, plotchar

NAME

getprefix & setprefix - get & set current default prefix

SYNOPSIS

```
getprefix (bp)          /* ProDOS functions */
char *bp;

setprefix (bp)
char *bp;
```

DESCRIPTION

getprefix returns the current default prefix in the buffer pointed at by *bp.* The buffer should be at least 64 bytes long.

setprefix sets the current default prefix to the character string pointed at by *bp.* This string can't contain more than 63 characters.

The prefix character string that is passed between a program and these functions is in C format (ie, it's a null-terminated string).

DIAGNOSTICS

If no error occurs, *getprefix* and *setprefix* return 0 as their value. If an error occurs, they set a code in the global int *errno* and return -1 as their value.

NAME

screen manipulation functions:
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

SYNOPSIS

scr_beep()

scr_bs()

scr_tab()

scr_lf()

scr_cursup()

scr_cursrt()

scr_cr()

scr_clear()

scr_home()

scr_eol()

scr_linsert()

scr_ldelete()

scr_cinsert()

scr_cdelete()

scr_curs(lin, col)
int lin, col;

DESCRIPTION

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.

These functions can be used in conjunction with the normal standard i/o and unbuffered i/o functions to display characters on the console.

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 positon 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.

NAME

setiob - pre-allocate unbuffered i/o blocks

SYNOPSIS

```
setiob(cnt)         /* ProDOS function */
int cnt;
```

DESCRIPTION

setiob preallocates *cnt* i/o blocks in the heap, setting pointers to them in the unbuffered i/o table.

When a file is opened, a 1K-byte block of memory that begins on a 1K-byte boundary must be allocated to the file; this block is used by ProDOS when it performs i/o operations on the file.

By default, i/o blocks are allocated only when a file is opened for which an unallocated i/o block can't be found. In this case, the i/o block is allocated by calling the *sbrk* function. When this is done, an area greater than 1K bytes may have to be allocated in order to place the i/o block on a 1K-byte boundary.

There are two reasons for using *setiob*:

* It causes needed i/o blocks to occupy a minimum amount of space.
* When used in conjunction with the *setbuf* function (which defines the i/o buffer to be used for standard i/o to a file), it allows a program to manage its own heap space, without the possibility of library functions taking a chunk out of the middle of the heap.

NAME

strchr & strrchr - find a character in a string

SYNOPSIS

```
char *strchr(s,c)
char *s; int c;

char *strrchr(s,c)
char *s; int c;
```

DESCRIPTION

strchr returns a pointer to the first occurrence of the character *c* in string *s*, or NULL if *c* isn't in the string. *c* can be the null character.

strrchr is like *strchr*, except that it returns a pointer to the last occurrence of *c* in *s*, rather than the first.

SEE ALSO

strchr and *strrchr* are the ANSI standard equivalents of the non-standard functions *index* and *rindex*, respectively. The only difference between them is that the *str...* functions can find a null character, while the *index* functions can't.

NAME

_system - issue ProDOS function call

SYNOPSIS

```
#include <sysfunc.h>        /* ProDOS function */

_system (func)
int func;
```

DESCRIPTION

_system issues the function call whose number is *func*, using as the parameter area the globally-accessible character array named *__sys__parm*.

DIAGNOSTICS

_system returns as its value the value that ProDOS returned in register A.

NAME

time, get_time, ctime, localtime, gmtime, asctime

SYNOPSIS

```
long time(tloc)        /* DOS functions */
long *tloc;

get_time(buf)
struct tm *buf;

char *ctime(clock)
long *clock;

#include "time.h"

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;
```

DESCRIPTION

time and *get_time* return the date and time, which they get from the operating system. The other functions convert the date and time, which are passed as arguments, to another format.

time returns the current date and time packed into a long int. If its argument *tloc* is non-null, the return value is also stored in the field pointed at by the argument. The format of the value returned by *time* is described below.

get_time returns the current date and time in the buffer pointed at by its argument, *buf*. The format of this buffer is described below.

ctime, *localtime*, and *gmtime* convert a date and time pointed at by their argument, which is in a format such as returned by *time*, to another format:

> *ctime* converts the time to a 26-character ASCII string of the form
>
> ```
> Mon Apr 30 10:04:52 1984\n\0
> ```
>
> *localtime* and *gmtime* unpack the date and time into a structure and return a pointer to it. The structure, named *tm*, is described below and defined in the header file *time.h*.
>
> *asctime* converts a date and time pointed at by its argument, which is in a structure such as returned by *localtime* and *gmtime*, to a 26-character ASCII string in the

same form as returned by *ctime.*

The long int returned by *time* and passed to *ctime*, *localtime*, and *gmtime* has the following form (bit 0 is the least significant bit in the field, bit 31 the most significant):

bits	*meaning*
00-07	minute
08-15	hour
16-20	day
21-24	month
25-31	year

The structure returned by *get_time*, *localtime* and *gmtime*, and passed to *asctime*, has the following format:

```
struct tm {
    short tm_sec;  /* seconds */
    short tm_min;  /* minutes */
    short tm_hour; /* hours */
    short tm_mday;/* day of the month */
    short tm_mon;  /* month */
    short tm_year; /* year since 1900 */
    short tm_wday;/* day of the week (0 = Sunday */
    short tm_yday; /* day of year */
    short tm_isdst; /* not used */
    short tm_hsec;  /* hundredths of seconds */
}
```

NAME

tmpfile - create a temporary file

SYNOPSIS

#include <stdio.h>
FILE *
tmpfile ()

DESCRIPTION

tmpfile creates a temporary file and opens it for standard i/o in update (w+) mode. *tmpfile* returns as its value the file's FILE pointer.

When the temporary file is closed, either because the program explicitly closes it or because the program terminates, the temporary file will automatically be deleted.

SEE ALSO

tmpnam, mktemp

NAME

tmpnam - create a name for a temporary file

SYNOPSIS

```
char *tmpnam (s)
char *s;
```

DESCRIPTION

tmpnam creates a character string that can be used as the name of a temporary file and returns as its value a pointer to the string. The generated string is not the name of an existing file.

s optionally points to an area into which the name will be generated. This must contain at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in *stdio.h.*

s can also be a NULL pointer. In this case, the name will be generated in an internal array. The contents of this array are destroyed each time *tmpnam* is called with a NULL argument.

The generated name is prefixed with the string that is associated with the symbol *P_tmpnam*; this symbol is defined in *stdio.h*. In the distribution version of *stdio.h*, *P_tmpnam* is a null string; this results in the generated name specifying a file that will be located in the current directory.

SEE ALSO

tmpfile, mktemp