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.

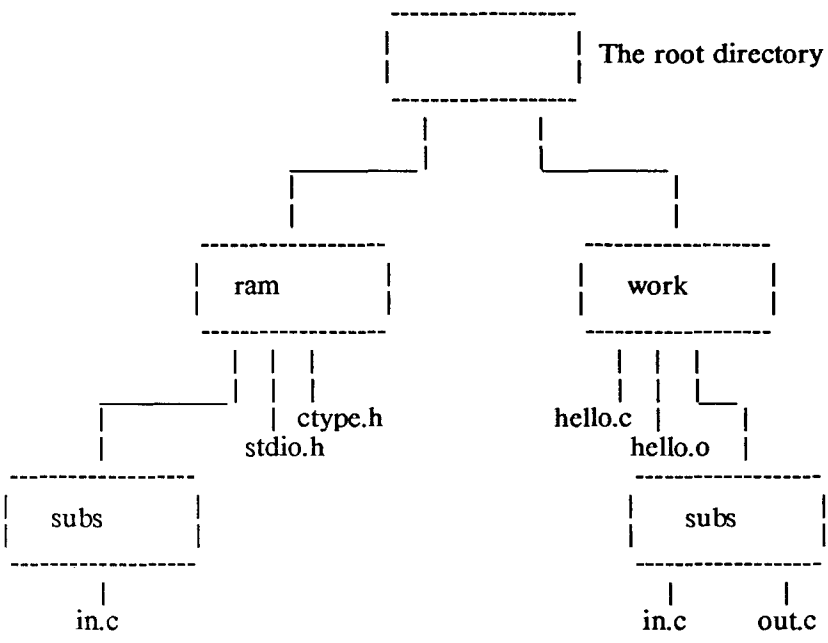

Figure 1: a sample file system

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