NitrOS-9 Operating System User's Guide

Version 03.02.01

NitrOS-9 Operating System User's GuideVersion 03.02.01

Revision History

Revision A December 2003
Updated with the changes made by Boisy Pitre

Table of Contents

Welcome to NitrOS-9! vi

1. Getting Started... 1

1.1. What You Need to Run NitrOS-9 1

1.1.1. Starting the System 1

1.1.2. In Case You Have Problems Starting NitrOS-9 1

1.1.3. A Quick Introduction to the Use of the Keyboard and Disks 2

1.1.4. Initial Explorations 2

1.2. Making a Backup of the System Disk 3

1.2.1. Formatting Blank Disks 3

1.2.2. Running the Backup Program 4

2. Basic Interactive Functions 5

2.1. Running Commands and Basic Shell Operation 5

2.1.1. Sending Output to the Printer 5

2.2. Shell Command Line Parameters 6

2.3. Some Common Command Formats 7

2.4. Using the Keyboard and Video Display 7

2.4.1. Video Display Functions 7

2.4.2. Keyboard Shift and Control Functions 8

2.4.3. Control Key Functions 8

3. The NitrOS-9 File System 10

3.1. Introduction to the Unified Input/Output System 10

3.2. Pathlists: How Paths Are Named 10

3.3. I/O Device Names 11

3.4. Multifile Devices And Directory Files 12

3.5. Creating and Using Directories 12

3.6. Deleting Directory Files 14

3.7. Additional Information About Directories 14

3.8. Using and Changing Working Directories 15

3.8.1. Automatic Selection of Working Directories 15

3.8.2. Changing Current Working Directories 16

3.8.3. Anonymous Directory Names 16

3.9. The File Security System 17

3.9.1. Examining and Changing File Attributes 17

3.10. Reading and Writing From Files 18

3.10.1. File Usage in NitrOS-9 18

3.10.2. Text Files 19

3.10.3. Random Access Data Files 19

3.10.4. Executable Program Module Files 19

3.10.5. Directory Files 20

3.10.6. Miscellaneous File Usage 20

3.11. Physical File Organization 20

4. Advanced Features of the Shell 22

4.1. A More Detailed Description Command Line Processing 22

4.2. Execution Modifiers 23

4.2.1. Alternate Memory Size Modifier 23

4.2.2. I/O Redirection Modifiers 23

4.3. Command Separators 24

4.3.1. Sequential Execution 24

4.3.2. Concurrent Execution 24

4.3.3. Pipes and Filters 25

4.4. Command Grouping 26

4.5. Built-in Shell Commands and Options 26

4.6. Shell Procedure Files 27

4.7. Error Reporting 28

4.8. Running Compiled Intermediate Code Programs 28

4.9. Setting Up Timesharing System Procedure Files 28

5. Multiprogramming and Memory Management 30
5.1. Processor Time Allocation and Timeslicing 30
5.2. Process States 31
5.3. Creation of New Processes 31
5.4. Basic Memory Management Functions 32
5.4.1. Loading Program Modules Into Memory 33
5.4.2. Loading Multiple Programs 34
5.4.3. Memory Fragmentation 34
6. Use of the System Disk 35
6.1. The OS9Boot File 35
6.2. The SYS Directory 35
6.3. The Startup File 36
6.4. The CMDS Directory 36
6.5. The DEFS Directory 36
6.6. Changing System Disks 36
6.7. Making New System Disks 37
7. System Command Descriptions 38
7.1. Formal Syntax Notation 38
7.2. Commands 38
ATTR 38
BACKUP 39
BINEX 41
BUILD 41
CHD/CHX 42
CMP 43
COBBLER 44
COPY 44
CPUTYPE 45
DATE 46
DCHECK 47
DEBUG 49
DED 50
DEL 51
DELDIR 52
DEVS 53
DMODE 53
DIR 54
DISASM 55
DISPLAY 56
DSAVE 56
DUMP 58
ECHO 59
EX 59
EXBIN 60
EXMODE 61
FORMAT 62
FREE 63
HELP 64
IDENT 65
INIZ 66
IRQS 67
KILL 67
LINK 68
LIST 69
LOAD 70
LOGIN 71
MAKDIR 72
MDIR 73
MERGE 74

MFREE....74
OS9GEN....75
PRINTERR....76
PROCS....77
PWD/PXD....78
RENAME....79
RUNB....79
SAVE....80
SETIME....80
SETPR....81
SHELL....82
SLEEP....83
TEE....84
TMODE....84
TSMON....86
TUNEPORT....87
UNLINK....88
VERIFY....89
XMODE....90
A. OS-9 Error Codes....92
A.1. Device Driver Errors....93
B. VDG Display System Functions....94
B.1. The Video Display Generator....94
B.2. Alpha Mode Display....94
B.3. Graphics Mode Display....95
B.4. Get Status Commands....97
C. Key Definitions With Hexadecimal Values....99

Welcome to NitrOS-9!

The foundation of a modern computer system is its *Operating System* or "OS". The OS is the master control program that interfaces all other software to the system's hardware. Some of the things it must do are performing input and output operations, coordinating memory use, and many other "housekeeping" functions. All other software - programming languages, applications programs, etc. - live in your computer's memory along with the OS and depend on it to communicate with you using the keyboard and display and to store and retrieve data on disks, etc. Because virtually all other software relies on the OS, your computer's performance depends on the capabilities and efficiency of its OS.

NitrOS-9 is an operating system for the TRS-80/Tandy Color Computer family which provides harmony to the chaos of multiple devices, memory management and application service requests. Its overall structure was based on the famous UNIX[1] operating system.

NitrOS-9 is 100% compatible with OS-9 Level One and OS-9 Level Two for the Color Computer. These products were manufactured by Tandy Corporation and sold through Radio Shack stores in the 1980s, and were purchased by many Color Computer users. Today, the Color Computer is no longer manufactured and those versions of OS-9 are no longer for sale. NitrOS-9 has evolved to be even better than these operating systems, and stands as the logical choice for those Color Computer owners who wish to use their CoCos today and tomorrow.

Some of the advanced NitrOS-9 features you'll learn about in this book are:

1. Multiuser/Multitasking Real-Time Operating System
2. Extensive support for structured, modular programming
3. Device-independent interrupt-driven input/output system
4. Multi-level directory file system
5. Fast Random-Access File System
6. Readily Expandable and Adaptable Design

If you don't know what some of these thing mean yet - don't worry. As you explore NitrOS-9 you'll soon learn how they enhance the capability of your Color Computer; and make it so much easier to use in almost any application.

NitrOS-9 has many commands and functions - definitely more than you can learn in an evening! The best way to become an NitrOS-9 expert is to study this manual carefully, section-by-section, taking time to try out each command or function. Because many functions affect others, you'll find this manual extensively cross-referenced so you can skip ahead to help you understand a new topic. Taking the time to study this book will certainly increase your knowledge and enjoyment of NitrOS-9.

But if you can't wait, at least read the rest of this chapter, scan the command descriptions in a later chapter, and have fun experimenting!

Notes

1. UNIX is an operating system designed by Bell Telephone Laboratories, which is becoming widely recognized as a standard for mini and micro operating systems because of its versatility and elegant structure.

Chapter 1. Getting Started...

1.1. What You Need to Run NitrOS-9

To use NitrOS-9 Level 1, you'll need the following:

- TRS-80/Tandy Color Computer or Color Computer 2 with 64K of RAM
- Disk Drive With Controller Cartridge

To use NitrOS-9 Level 2, you'll need the following:

- Tandy Color Computer 3 with 128K of RAM (512K recommended)
- Disk Drive With Controller Cartridge

NitrOS-9 is also ready to use the following optional equipment that you may have now or may obtain in the future:

- Additional Floppy Disk Drives
- SCSI or IDE Hard Drives
- Printers and Modems
- Additional Serial Ports
- Joysticks and Mice
- Other NitrOS-9 Compatible Languages and Software

1.1.1. Starting the System

To start up NitrOS-9 from a floppy drive, follow these steps:

1. Turn the Color Computer and disk drive(s) on. You should see the usual BASIC greeting message on the screen.
2. Insert the NitrOS-9 System Disk in drive zero and close the door.
3. Type "DOS". After a few seconds of disk activity you should see a screen with the words "NITROS9 BOOT".
4. NitrOS-9 will then begin its "bootstrap" loading process, which involves a number of seconds of disk activity. When the system startup has finished, a message followed by the shell prompt will be displayed.

1.1.2. In Case You Have Problems Starting NitrOS-9

- If BASIC gives an error message after you type "DOS", remove the disk, turn the computer off and on, then try again. If this repeatedly fails your NitrOS-9 diskette may be bad.
- Did you remember to turn the disk drive power switch on?
- Does your Color Computer meet the minimum RAM requirements? This is a must!
- If your Color Computer doesn't seem to understand the DOS command, your controller has DOS 1.0. You will need to upgrade to DOS 1.1.

- If the "NITROS9 BOOT" message is displayed but nothing else happens, you may have a corrupted system disk. Hopefully you did make a backup!

1.1.3. A Quick Introduction to the Use of the Keyboard and Disks

For now, the only special keys on the keyboard of interest are the SHIFT key which works like a typewriter shift key; the ENTER key which you always use after typing a command or response to NitrOS-9; and the <- left arrow key which you can use to erase typing mistakes.

Your main disk drive is known to to NitrOS-9 as "/D0" and is often called "drive zero". If you have a second disk drive (drive one), NitrOS-9 recognizes it as "/D1". Why would anybody put a "/" in a name? Because all input and output devices have names like files, and names that start with "/" are always device names.

1.1.4. Initial Explorations

When NitrOS-9 first starts up, it will display a welcoming message, and then ask you to enter the date and time. This allows NitrOS-9 to keep track of the date and time of creation of new files and disks. Enter the current date and time in the format requested like this:

```
            yyyy/mm/dd hh:mm:ss
    Time ?  2004 01 01 14 20
```

In the example above, the date entered was January 1, 2004. NitrOS-9 uses 24-hour time so the date entered was 1420 hours or 2:20 PM. Next, NitrOS-9 will print the shell prompt to let you know it is ready for you to type in a command.

Now you're ready to try some commands. A good first command to try is **dir** (for "*dir*ectory"). This will display a list of the files on the System Disk. Just type:

```
dir
```

followed by a "return". NitrOS-9 should respond with a listing of file names which should look something like this:

```
  Directory of .
OS9Boot         CMDS            SYS             DEFS            sysgo
startup
```

The file `OS9Boot` contains the NitrOS-9 program in 6809 machine language, which was loaded into memory during the bootstrap operation.

The file `sysgo` is only located on the NitrOS-9 Level 2 System Disk. It is the first program run on the system, and kick-starts the initial application (usually, the shell).

The file `startup` is a "command file" which is automatically run when the system starts up, and has the commands that printed the welcoming message and asked for the time. Later, You may want to replace this startup file with your own customized version after you are more familiar with NitrOS-9. Do you want to see the contents of this file? If so, just type

```
list startup
```

As you can see, the **list** command displays the contents of files that contain text (alphabetic characters). Some files like the `OS9Boot` file contain binary data such as machine language programs. These files are called "binary files", and attempts to list them will result in a jumbled, meaningless display. On the other hand, NitrOS-9 will complain mightily if you try to run a text file as a program!

As you may have surmised by now, the way you ask NitrOS-9 to run a program or command (they're really the same thing) is to simply type its name. Some commands like **list** require one or more names of files or options. If so, they are typed on the same line using spaces to separate each item.

But where did the **list** and **dir** programs come from? There are really more files on the disk than you suspect. The **dir** command showed you what is the disk's *root directory* - so named because the NitrOS-9 file system resembles a tree. Growing out of the root directory are three "branches" - files which are additional directories of file names instead of programs or data. They in turn can have even more "branches" - ad infinitum. If you draw a map on paper of how this works it does look like a tree. The directory files on your system disk are called `CMDS`, `SYS`, and `DEFS`. The file `CMDS` is a directory that consists of all the system commands such as **dir**, **list**, **format**, etc. To see the files contained in this directory, enter:

```
dir cmds
```

which tells **dir** to show files on the directory file `CMDS` instead of the root directory. After you type this you should see a long list of file names. These are the complete set of command programs that come with NitrOS-9 and perform a myriad of functions. Chapter Seven explains each one in detail. The **dir** command also has a handy option to display the `CMDS` directory with less typing:

```
dir -x
```

Whenever you want a list of available commands you can use this so you don't have to look it up in the book. The **dir** command has options which can give you more detailed information about each file.

1.2. Making a Backup of the System Disk

Before getting too much more involved in further experimentation, NOW is the time to make one or more exact copies of your System Disk in case some misfortune befalls your one and only master System Disk. Making a backup involves two steps: formatting a blank disk and running a backup program.

1.2.1. Formatting Blank Disks

Before the actual backup procedure can be done (or any fresh diskette is used for any purpose), the blank disk which is to become the backup disk must be initialized by NitrOS-9's **format** command.

IF YOU HAVE ONLY ONE DISK DRIVE you have to be extra careful not to accidentally FORMAT your system disk. Type:

```
format /d0
```

and when you see the message

```
COLOR COMPUTER FORMATTER
Formatting drive /d0
y (yes) or n (no)
Ready?
```

immediately remove your system disk and insert a blank disk *before* you type "Y". IF YOU HAVE TWO DISK DRIVES place the blank disk in drive one and type:

```
format /d1
```

WHEN THE BLANK DISK IS IN THE RIGHT PLACE, type "Y", then "ENTER". This initiates the formatting process. IF THE CORRECT DEVICE NAME (/D1) IS NOT DISPLAYED: TYPE N RIGHT NOW and start over, OR YOU MAY ERASE your System Disk.

When you are asked for a disk name, type any letter, then ENTER. The name you give is not important. If you have only one drive, replace the system disk after the FORMAT program has finished. If the FORMAT program reported any errors, try again. Disks used for backups can't have any errors. You're now ready to run the **backup** program.

It takes several minutes for the FORMAT program to run. During its second phase the hexadecimal number of each track will be displayed as it is checked for bad sectors. If any are found an error message for each bad sector is given.

1.2.2. Running the Backup Program

The **backup** program makes an exact duplicate of a disk. It can be used even if you only have one disk drive.

IF YOU HAVE ONE DRIVE type

```
backup /d0 #32k
```

The **backup** program will prompt you to alternately insert the source disk (the system disk) and the destination disk (the freshly formatted disk).

IF YOU HAVE TWO DRIVES type

```
backup #32K
```

The **backup** program will respond with

```
Ready to backup from /d0 to /d1?
```

Now enter Y for yes. It will then ask:

```
X is being scratched
Ok ?:
```

Answer "Y" for yes again, and the backup process should begin.

The **backup** command has two phases: the first phase copies everything from drive zero to drive one checking for errors while reading from the master but not for "write" errors. The second phase is the "verify" pass which makes sure everything was copied onto the new disk correctly. If any errors are reported during the first (copy) pass, there is a problem with the master disk or its drive. If errors occur during the second (verify) pass, there is a problem with the new disk and the **backup** program should be run again. If **backup** repeatedly fails on the second pass, reformat the disk and try to **backup** again. If **backup** fails again, the disk is physically defective.

After you have made your backup disk, try turning the Color Computer off and restarting the system with the copy you just made. If it works OK, store it in a safe place in case it is needed later. You should always have a backup copy of your system disk and all other important disks.

Chapter 2. Basic Interactive Functions

2.1. Running Commands and Basic Shell Operation

The **shell** is a the part of NitrOS-9 that accepts commands from your keyboard. It was designed to provide a convenient, flexible, and easy-to-use interface between you and the powerful functions of the operating system. The shell is automatically entered after NitrOS-9 is started up. You can tell when the shell is waiting for input because it displays the shell prompt. This prompt indicates that the shell is active and awaiting a command from your keyboard. It makes no difference whether you use upper-case letters, lower-case letters, or a combination of both because NitrOS-9 matches letters of either case.

The command line always begins with a name of a program which can be:

- The name of a machine language program on disk
- The name of a machine language program already in memory
- The name of an executable program compiled by a high-level language such as Basic09, Pascal, Cobol, etc.
- The name of a procedure file

If you're a beginner, you will almost always use the first case, which causes the program to be automatically loaded from the `CMDS` directory and run.

When processing the command line, the shell searches for a program having the name specified in the following sequence:

1. If the program named is already in memory, it is run.
2. The "execution directory", usually `CMDS`, is searched. If a file having the name given is found, it is loaded and run.
3. The user's "data directory" is searched. If a file having the name given is found, it is processed as a "procedure file" which means that the file is assumed to contain one or more command lines which are processed by the shell in the same manner as if they had manually typed in one by one.

Mention is made above of the "data directory" and the "execution directory". At all times each user is associated with two file directories. A more detailed explanation of directories is presented later. The execution directory (usually `CMDS`) includes files which are executable programs.

The name given in the command line may be optionally followed by one or more "parameters" which are passed to the program called by the shell.

For example, in the command line:

```
list file1
```

the program name is **list**, and the parameter passed to it is FILE1.

A command line may also include one or more "modifiers" which are specifications used by the shell to alter the program's standard input/output files or memory assignments.

2.1.1. Sending Output to the Printer

Normally, most commands and programs display output on the Color Computer video display. The output of these programs can alternatively be printed by specifying output redirection on the command line. This is done by including the following modifier to at the end of any command line:

```
>/p
```

The ">" character tells the shell to redirect output (See Section 4.3.2) to the printer using the Color Computer's printer port, which has the device name "/p" (See Section 3.3). For example, to redirect the output of the **dir** command to the printer, enter:

```
dir >/p
```

The **xmode** command can be used to set the printer port's operating mode such as auto line feed, etc. For example, to examine the printer's current settings, type:

```
xmode /p
```

To change any of these type XMODE followed by the new value. For example, to set the printer port for automatic line feeds at the end of every line, enter:

```
xmode /p lf
```

2.2. Shell Command Line Parameters

Parameters are generally used to either specify file name(s) or to select options to be used by the program specified in the command line given to the shell. Parameters are separated from the command name and from each other by space characters (hence parameters and options cannot themselves include spaces). Each command program supplied with NitrOS-9 has an individual description in the last section of this manual which describe the correct usage of the parameters of each command.

For example, the **list** program is used to display the contents of a text file on your display. It is necessary to tell to the **list** program which file it is to be displayed, therefore, the name of the desired file is given as a parameter in the command line. For example, to list the file called startup (the system initialization procedure file), you enter the command line:

```
list startup
```

Some commands have two parameters. For example, the **copy** command is used to make an exact copy of a file. It requires two parameters: The name of the file to be copied and the name of the file which is to be the copy, for example:

```
copy startup newstartup
```

Other commands have parameters which select options. For example:

```
dir
```

shows the names of the files in the user's data directory. Normally it simply lists the file names only, but if the "-e" (for *e*ntire) option is given, it will also give complete statistics for each file such as the date and time created, size, security codes, etc. To do so enter:

```
dir -e
```

The **dir** command also can accept a file name as a parameter which specifies a directory file other than the (default) data directory. For example, to list file names in the directory sys , type:

```
dir sys
```

It is also possible to specify both a directory name parameter and the -e option, such as:

```
dir sys -e
```

giving file names and complete statistics.

2.3. Some Common Command Formats

This section is a summary of some commands commonly used by new or casual NitrOS-9 users, and some common formats. Each command is followed by an example. Refer to the individual command descriptions later int his book for more detailed information and additional examples. Parameters or options shown in brackets are optional. Whenever a command references a directory file name, the file *must* be a directory file.

```
CHD filename                                chd DATA.DIR
```

Changes the current *data* working directory to the *directory* file specified.

```
COPY filename1 filename2                    copy oldfile newfile
```

Creates filename2 as a new file, then copies all data from "filename1" to it. "filename1" is not affected.

```
DEL filename                                del oldstuff
```

Deletes (destroys) the file specified.

```
DIR [filename] [-e] [-x]                    dir myfiles -e
```

List names of files contained in a directory. If the "-x" option is used the files in the current *execution* directory are listed, othervise, if no directory name is given, the current *data* directory will be listed. The "-e" option selects the long format which shows detailed information about each file.

```
FREE devicename                             free /d1
```

Shows how much free space remains on the disk whose name is given.

```
LIST filename                               list script
```

Displays the (text) contents of the file on the terminal.

```
MAKDIR filename                             makdir NEWFILES
```

Creates a new directory file using the name given. Often followed by a **chd** command to make it the new working data directory.

```
RENAME filename1 filename2                  rename zip zap
```

Changes the name of filename1 to filename2.

2.4. Using the Keyboard and Video Display

NitrOS-9 has many features to expand the capability of the Color Computer keyboard and video display. The video display has screen pause, upper/lower case, and graphics functions. The keyboard can generate all ASCII characters and has a type-ahead feature that permits you to enter data before requested by a program (except if the disk is running because interrupts are temporarily disabled). Appendix B of this manual is a list of the characters and codes that can be generated from the keyboard. The keyboard/video display can be used as a file known by the name "/TERM".

2.4.1. Video Display Functions

Most Color Computers use reverse video (green letters in a black box) to represent lower-case letters. Normally they are not used, so you have to turn them on if you want to use them with the command:

```
tmode -upc
```

However, the Color Computer 3 and certain models of the Color Computer 2 can also do true lowercase on the 32x16 video display. To see if your Color Computer can do true lowercase, type the following command:

```
xmode /term typ=1; display e
```

If your Color Computer cannot do true lower case, your screen will show graphics garbage.

The screen pause feature stops programs after 16 lines have been displayed. Output will continue if you hit any key. Normally this feature is on. It can be turned on or off with the **tmode** command as follows:

```
tmode -pause              turns pause mode off
tmode pause               turns pause mode on
```

The display system also has a complete set of commands to emulate commercial data terminals, plus a complete set of graphics commands. These are described in detail in Appendix C.

2.4.2. Keyboard Shift and Control Functions

Two keys are used in combination with other keys to change their meaning. The SHIFT KEY selects between upper case and lower case letters or punctuation, and the **CLEAR** key can be used to generate control characters.

The keyboard has a shift lock function similar to a typewriter's, which is normally "locked". The keyboard's shift lock may be reversed by depressing the control key and **0** keys simultaneously. The shift lock only affects the letter (A-Z) keys. When the keyboard is locked, these keys generate upper case letters, and lower case only if the SHIFT key is depressed. When the keyboard is unlocked, the reverse is true, e.g., lower case letters are generated unless the SHIFT key is depressed at the same time as a letter key.

2.4.3. Control Key Functions

There are a number of useful control functions that can be generated from the keyboard. Many of these functions use "control keys" which are generated by simultaneously depressing the **CLEAR** key plus some other key. For example, to generate the character for **CONTROL D** press the **CLEAR** and **D** keys at the same time.

CONTROL+A

Repeat previous input line. The last line entered will be redisplayed but *not* processed, with the cursor positioned at the end of the line. You may hit return to enter the line, or edit the line by backspacing, typing over characters to correct them, and entering control A again to redisplay the edited line.

CONTROL+D

Redisplay present input on next line.

CONTROL+W

Display Wait - This will temporarily halt output to the display so the screen can be read before the data scrolls off. Output is resumed when any other key is hit.

CONTROL+0

Shift lock. Reverses present shift lock state.

BREAK (or **CONTROL+E**)

Program abort - Stops the current running program

SHIFT+BREAK (or **CONTROL+C**)

Interrupt - Reactivates Shell while keeping program running as background task.

CONTROL+BREAK (**ESCAPE**)

End-of-File - This key is used to send an end-of-file to programs that read input from the terminal in place of a disk or tape file. It must be the first character on the line in order for it to be recognized.

LEFT ARROW (OR **CONTROL+H**)

Backspace - erase previous character

SHIFT+LEFT ARROW (or **CONTROL+X**)

Line Delete - erases the entire current line.

Chapter 3. The NitrOS-9 File System

3.1. Introduction to the Unified Input/Output System

NitrOS-9 has a unified input/output system in which data transfers to ALL I/O devices are performed in almost exactly the same manner, regardless of the particular hardware devices involved. It may seem that the different operational characteristics of the I/O devices might make this difficult. After all, line printers and disk drives behave much differently. However, these differences can mostly be overcome by defining a set of standardized *logical functions* for all devices and by making all I/O devices conform to these conventions, using software routines to eliminate hardware dependencies wherever possible. This produces a much simpler and more versatile input/output system.

NitrOS-9's unified I/O system is based upon logical entities called "I/O paths". Paths are analogous to "software I/O channels" which can be routed from a program to a mass-storage file, any other I/O device, or even another program. Another way to say the same thing is that paths are files, and all I/O devices behave as files.

Data transferred through paths may be processed by NitrOS-9 to conform to the hardware requirements of the specific I/O device involved. Data transfers can be either bidirectional (read/write) or unidirectional (read only or write only), depending on the device and/or how the path was established.

Data transferred through a path is considered to be a stream of 8-bit binary bytes that have no specific type or value: what the data actually represents depends on how it is used by each program. This is important because it means that NitrOS-9 does not require data to have any special format or meaning.

Some of the advantages of the unified I/O system are:

- Programs will operate correctly regardless of the particular I/O devices selected and used when the program is actually executed.
- Programs are highly portable from one computer to another, even when the computers have different kinds of I/O devices.
- I/O can be redirected to alternate files or devices when the program is run, without having to alter the program.
- New or special device driver routines can easily be created and installed by the user.

3.2. Pathlists: How Paths Are Named

Whenever a path is established (or "opened"), NitrOS-9 must be given a description of the "routing" of the path. This description is given in the form of a character string called a "pathlist". It specifies a particular mass-storage file, directory file, or any other I/O device. NitrOS-9 "pathlists" are similar to "filenames" used by other operating systems.

The name "pathlist" is used instead of "pathname" or "filename" because in many cases it is a list consisting of more than one name to specify a particular I/O device or file. In order to convey all the information required, a pathlist may include a device name, one or more directory file names and a data file name. Each name within a pathlist is separated by slash "/" characters.

Names are used to describe three kinds of things:

- Names of Physical I/O Devices
- Names of Regular Files

- Names of Directory Files

Names can have one to 29 characters, all of which are used for matching. They may be composed of any combination of the following characters:

```
uppercase letters: A - Z
lowercase letters: a - z
decimal digits: 0 - 9
underscore: _
period: . (cannot be the first character)
```

Here are examples of *legal* names:

raw.data.2	projectreview.backup
reconciliation.report	X042953
RJJones	22search.bin

Here are examples of *illegal* names:

max*min	(* is not a legal character)
.data	(does not start with a letter)
open orders	(cannot contain a space)
this.name.obviously.has.more.than.29.characters	(too long)

3.3. I/O Device Names

Each physical input/output device supported by the system must have a unique name. The actual names used are defined when the system is set up and cannot be changed while the system is running. The device names used for the Color Computer are:

TERM	Video display/keyboard
P	Printer port
D0	Disk drive unit zero
D1	Disk drive unit one
PIPE	Pipes

Device names may only be used as the first name of a pathlist, and must be preceded by a slash "/" character to indicate that the name is that of an I/O device. If the device is not a disk or similar device the device name is the only name allowed. This is true for devices such as terminals, printers, etc. Some examples of of pathlists that refer to I/O devices are:

```
/TERM
/P
/D1
```

I/O device names are actually the names of the "device descriptor modules" kept by NitrOS-9 in an internal data structure called the "module directory" (See the NitrOS-9 System Programmer's manual for more information about device driver and descriptor modules). This directory is automatically set up during NitrOS-9's system start up

sequence, and updated as modules are added or deleted while the system is running.

3.4. Multifile Devices And Directory Files

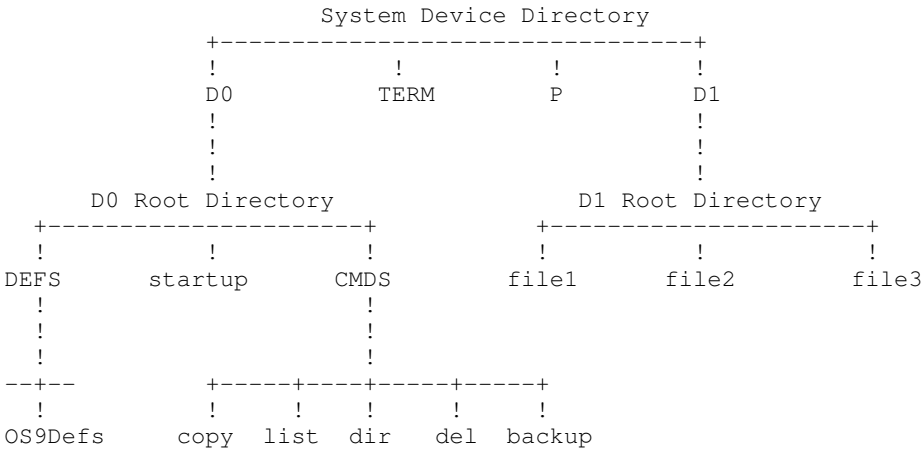
Multifile devices are mass storage devices (usually disk systems) that store data organized into separate logical entities called "files". Each file has a name which is entered in a directory file. Every multifile device has a master directory (called the "root directory") that includes the names of the files and sub-directories stored on the device. The root directory is created automatically when the disk is initialized by the **format** command.

Pathlists that refer to multifile devices may have more than one name. For example, to refer to the file "mouse" whose name appears in the root directory of device "D1" (disk drive one) the following pathlist is used:

/d1/mouse

When NitrOS-9 is asked to create a path, it uses the names in the pathlist sequentially from left to right to search various directories to obtain the necessary routing information. These directories are organized as a tree-structured hierarchy. The highest-level directory is called the "device directory", which contains names and linkages to all the I/O devices on a given system. If any of the devices are of a multifile type they each have a root directory, which is the next-highest level.

The diagram below is a simplified file system tree of a typical NitrOS-9 system disk. Note that device and directory names are capitalized and ordinary file names are not. This is a customary (but not mandatory) practice which allows you to easily identify directory files using the short form of the **dir** command.



The device names in this example system are "TERM", "P", "D0" and "D1". The root directory of device "D0" includes two directory files, `DEFS` and `CMDS`, and one ordinary file "startup". Notice that device "D1" has in its root directory three ordinary files. In order to access the file "file2" on device "d1", a pathlist having two names must be used:

```
list /d1/file2
```

To construct a pathlist to access the file "dir" on device "d0" it is necessary to include in the pathlist the name of the intermediate directory file `CMDS`. For example, to copy this file requires a pathlist having three names to describe the "from" file:

```
copy /d0/cmds/dir temp
```

3.5. Creating and Using Directories

It is possible to create a virtually unlimited number of levels of directories on a mass storage device using the **makdir** command. Directories are a special type of file (see Section 3.9.1). They can be processed by the same I/O functions used to access regular files which makes directory-related processing fairly simple.

To demonstrate how directories work, assume that the disk in drive one ("d1") has been freshly formatted so that it has a root directory only. The **build** command can be used to create a text file on "d1". The **build** command will print out "?" as a prompt to indicate that it is waiting for a text line to be entered. It will place each line into the text file until an empty line with only a carriage return is entered, as shown below:

```
OS9: build /d1/file1
? This is the first file that
? we created.
? [ENTER]
```

The **dir** command will now indicate the existence of the new file:

```
OS9: dir /d1

   Directory of /d1  15:45:29
file1
```

The **list** command can be used to display the text stored in the file:

```
OS9: list /d1/file1

This is the first file
that we created.
```

The **build** command again is again used to create two more text files:

```
OS9: build /d1/file2
? This is the second file
? that we created.
? [ENTER]

OS9: build /d1/file3
? This is another file.
? [ENTER]
```

The **dir** command will now show three file names:

```
OS9: dir /d1
   Directory of /D1  15:52:29
file1           file2           file3
```

To make a new directory in this directory, the **makdir** command is used. The new directory will be called `NEWDIR`. Notice that throughout this manual directory names are always capitalized. This is *not* a requirement of NitrOS-9 (see Section 3.2) . Rather, it is a practice popular with many NitrOS-9 users because it allows easy identification of directory files at all times (assuming all other file names use lower-case letters).

```
OS9: makdir /D1/NEWDIR
```

The directory file `NEWDIR` is now a file listed in D1's root directory:

```
OS9: dir /D1

   Directory of /D1  16:04:31
file1           file2           file3          NEWDIR
```

Now we will create a new file and put in the new directory, using the **copy** command to duplicate `file1`:

```
OS9: copy /d1/file1 /d1/newdir/file1.copy
```

Observe that the second pathlist now has three names: the name of the root directory ("D1"), the name of the next lower directory (`NEWDIR`), then the actual file name (`file1.copy`). Here's what the directories look like now:

```
          D1 Root Directory
    +---------+--------+--------+
    !         !        !        !
  NEWDIR    file1    file2    file3
    !
    !
file1.copy
```

The **dir** command can now show the files in the new directory:

```
OS9: dir /D1/NEWDIR

   Directory of /D1/NEWDIR
file1.copy
```

It is possible to use **makdir** to create additional new directories within `NEWDIR`, and so on, limited only by available disk space.

3.6. Deleting Directory Files

The **del** command cannot be used to directly delete a directory file. If a directory file that still contained file names were to be deleted, NitrOS-9 would have no way to access the files or to return their storage to the unallocated storage pool. Therefore, the following sequence must be performed to delete a directory file:

1. All file names in the directory must be deleted.

2. The **attr** command is used to turn off the files directory attrribute (-d option), making it an ordinary file (see Section 3.9).

3. The file may now be deleted using the **del** command.

A simpler alternative is to use the **deldir** command to automatically perform all these steps for you.

3.7. Additional Information About Directories

The NitrOS-9 directory system is very useful because it allows each user to privately organize files as desired (by project, function, etc.), without affecting other files or other user's files. Another advantage of the hierarchical directory system is that files with identical names can be kept on the same device as long as the names are in different directories. For example, you can have a set of test files to check out a program using the same file names as the program's actual working files. You can then run the program with test data or actual data simply by switching directories.

Here are some important characteristics relating to use of directory files:

- Directories have the same ownership and security attributes and rules as regular files.
- The name of a given file appears in exactly one directory.
- Files can only be added to directories when they are created.

- A file and the directory in which its name is kept must reside on the same device.

3.8. Using and Changing Working Directories

Each program (process) has two "working directories" associated with it at all times: a "data directory" and an "execution directory". The working directory mechanism allows the name searching involved in pathlist processing to start at any level (subtree) of the file system hierarchy. Any directory that the user has permission to access (see Section 3.9) can be made a working directory.

The rules used to determine whether pathlists refer to the current working directory or not are simple:

---> When the first character of a pathlist IS a "/", processing of the pathlist starts at the device directory, e.g., the first name MUST be a device name.

---> When the first character of a pathlist IS NOT a "/", processing of the pathlist starts at the current working directory.

Notice that pathlists starting with a "/" *must* be complete, in other words, they must have all names required to trace the pathlist from the device directory down through all intermediate directories (if any). For example:

/d2/JOE/WORKINGFILES/testresults

On the other hand, use of the current working directory allows all names in the file hierarchy tree to be implied instead of explicitly given. This not only makes pathlists shorter, but allows NitrOS-9 to locate files faster because (typically) fewer directories need be searched. For example, if the current working directory is `/D1/PETE/GAMES` and a pathlist is given such as:

baseball

the actual pathlist *implied* is:

/D1/PETE/GAMES/baseball

Pathlists using working directories can also specify additional lower-level directories. Referring to the example above, the pathlist:

ACTION/racing

implies the complete pathlist:

/D1/PETE/GAMES/ACTION/racing

3.8.1. Automatic Selection of Working Directories

Recall that two working directories are referred to as the "current execution directory" and the "current data directory". The reason two working directories are maintained is so that files containing *programs* can be organized in different directories than files containing *data*. NitrOS-9 automatically selects either working directory, depending on the usage of the pathlist:

---> NitrOS-9 will search the execution directory when it attempts to load files into memory assumed to be executable programs. This means that programs to be run as commands or loaded into memory must be in the current execution directory.

---> The data directory is used for all other file references (such as text files, etc.)

Immediately after startup, NitrOS-9 will set the data directory to be (the root directory of) the system disk drive (usually "D0"), and the working directory to be a directory called `cmds` on the same drive (`/D0/cmds`). On timesharing systems, the **login**

command selects the initial execution and data directories to the file names specified in each user's information record stored in the system password file(ref. Section 5.4.2).

Here is an example of a **shell** command statement using the default working directory notation, and its equivalent expansion:

```
copy file1 file2
```

If the current execution directory is `/D0/CMDS` and the current data directory is `/D0/JONES`, the same command, fully expanded to show complete pathlists implied is:

```
OS9: /D0/CMDS/copy /D0/JONES/file1 /D0/JONES/file2
```

Notice that the first pathlist **copy** expands to the current working directory pathlist because it is assumed to be an executable program but the two other file names expand using the data directory because they are not assumed to be executable.

3.8.2. Changing Current Working Directories

The built-in shell commands **chd** and **chx** can be used to independently change the current working data and execution directories, respectively. These command names must be followed by a pathlist that describes the new directory file. You must have permission to access the directory according to normal file security rules. Here are some examples:

```
OS9: chd /D1/MY.DATAFILES

OS9: chx /D0/TESTPROGRAMS
```

When using the **chd** or **chx** commands, pathlists work the same as they do for regular files, except for the last name in the pathlist must be a directory name. If the pathlist begins with a "/" , NitrOS-9 will begin searching in the device directory for the new working directory, otherwise searching will begin with the present directory. For example, the following sequence of commands set the working directory to the same file:

```
OS9: CHD /D1/SARAH
OS9: CHD PROJECT1

OS9: CHD /D1/SARAH/PROJECT1    (same effect as above)
```

3.8.3. Anonymous Directory Names

Sometimes is useful to be able to refer to the current directory or the next higher-level directory, but its name (full pathlist) may not be known. Because of this, special "name substitutes" are available. They are:

. refers to the present working directory

. . refers to the directory that contains the name of the present directory (e.g., the next highest level directory)

. . . refers to directory two levels up, and so on

These can be used in place of pathlists and/or the first name in a pathlist. Here are some examples:

OS9: dir . lists file names in the working data directory

OS9: dir ..	lists names in the working data directory's parent directory.
OS9: del ../temp	deletes the file `temp` from the working data directory's parent directory.

The substitute names refer to either the execution or data directories, depending on the context in which they are used. For example, if `..` is used in a pathlist of a file which will be loaded and/or executed, it will represent the parent directory of the execution directory. Likewise, if `.` is used in a pathlist describing a program's input file, it will represent the current data directory.

3.9. The File Security System

Every file (including directory files) has properties called *ownership* and *attributes* which determine who may access the file and how it many be used.

NitrOS-9 automatically stores with each file the user number associated with the process that created it. This user is considered to be the "owner" of the file.

Usage and security functions are based on "attributes", which define how and by whom the file can be accessed. There are a total of seven attributes, each of which can be turned "off" or "on" independently. The "d" attribute is used to indicate (when on) that the file is a directory file. The other six attributes control whether the file can be read, written to, or executed, by either the owner or by the "public" (all other users). Specifically, these six attributes are:

WRITE PERMISSION FOR OWNER: If on, the owner may write to the file or delete it. This permission can be used to protect important files from accidental deletion or modification.

READ PERMISSION FOR OWNER: If on, the owner is allowed to read from the file. This can be used to prevent "binary" files from being used as "text" files

EXECUTE PERMISSION FOR OWNER: If on, the owner can load the file into memory and execute it. Note that the file *must* contain one or more valid NitrOS-9 format memory modules in order to actually load

The following "public permissions" work the same way as the "owner permissions" above but are applied to processes having DIFFERENT user numbers than the file's owner.

WRITE PERMISSION FOR PUBLIC: If on, any other user may write to or delete the file.

READ PERMISSION FOR PUBLIC: If on, any other user may read (and possibly copy) the file.

EXECUTE PERMISSION FOR PUBLIC: If on, any other user may execute the file.

For example, if a particular file had all permissions on except "write permit to public" and "read permit to public", the owner would have unrestricted access to the file, but other users could execute it, but not read, copy, delete, or alter it.

3.9.1. Examining and Changing File Attributes

The **dir** command may be used to examine the security permissions of the files in any particular directory when the "e" option is used. An example using the **dir e** command to show the detailed attributes of the files in the current working directory is:

```
Directory of .   2003/03/04 10:20
```

```
Owner  Last Modified    Attributes Sector Bytecount Name
----- ----------------- ---------- ------ --------- ----
   1  2002/05/29 14:02   --e--e-r      47        42 file1
   0  2002/10/12 02:15   ---wr-wr      48        43 file2
   3  2002/04/29 23:35   -s----wr      51        22 file3
   1  2003/01/06 16:19   d-ewrewr      6D       800 NEWDIR
```

This display is fairly self-explanatory. The "attributes" column shows which attributes are currently on by the presence or absence of associated characters in the following format:

dsewrewr

The character positions correspond to from left to right: directory; sharable; public execute; public write; public read; owner execute; owner write; owner read. The **attr** command is used to examine or change a file's attributes. Typing **attr** followed by a file name will result in the present attributes to be displayed, for example:

```
OS9: attr file2
-s-wr-ewr
```

If the command is used with a list of one or more attribute abbreviations, the file's attributes will be changed accordingly (if legal). For example, the command:

```
OS9: attr file2 pw pr -e -pe
```

enables public write and public read permissions and removes execute permission for both the owner and the public.

The "directory" attribute behaves somewhat differently than the read, write, and execute permissions. This is because it would be quite dangerous to be able to change directory files to normal files, and creation of a directory requires special initialization. Therefore, the **attr** command *cannot* be used to turn the directory (d) attribute on (only **makdir** can), and can be used to turn it off *only* if the directory is empty.

3.10. Reading and Writing From Files

A single file type and format is used for all mass storage files. Files store an ordered sequence of 8-bit bytes. NitrOS-9 is not usually sensitive to the contents of files for most functions. A given file may store a machine language program, characters of text, or almost anything else. Data is written to and read from files exactly as given. The file can be any size from zero up to the maximum capacity of the storage device, and can be expanded or shortened as desired.

When a file is created or opened a "file pointer" is established for it. Bytes within the file are addressed like memory, and the file pointer holds the "address" of the next byte in the file to be written to or read from. The NitrOS-9 "read" and "write" service functions always update the pointer as data transfers are performed. Therefore, successive read or write operations will perform sequential data transfers.

Any part of a file can also be read or written in non-sequential order by using a function called "seek" to reposition the file pointer to any byte address in the file. This is used when random access of the data is desired.

To expand a file, you can simply write past the previous end of the file. Reading up to the last byte of a file will cause the next "read" request to return an end-of-file status.

3.10.1. File Usage in NitrOS-9

Even though there is physically only one type of file, the logical usage of files in NitrOS-9 covers a broad spectrum. Because all NitrOS-9 files have the same physical

type, commands such as **copy**, **del**, etc., can be used with any file regardless of its logical usage. Similarly, a particular file can be treated as having a different logical usage at different times by different programs. The main usage of files covered in this section are:

TEXT
RANDOM ACCESS DATA
EXECUTABLE PROGRAM MODULES
DIRECTORIES
MISCELLANEOUS

3.10.2. Text Files

These files contain variable-length sequences ("lines") of ASCII characters. Each line is terminated by a carriage return character. Text files are used for program source code, procedure files, messages, documentation, etc. The Text Editor operates on this file format.

Text files are usually read sequentially, and are supported by almost all high-level languages (such as BASIC09 READ and WRITE statements). Even though is is possible to randomly access data at any location within a text file, it is rarely done in practice because each line is variable length and it is hard to locate the beginning of each line without actually reading the data to locate carriage return characters.

The content of text files may be examined using the **list** command.

3.10.3. Random Access Data Files

Random-access data files are created and used primarily from within high-level languages such as Basic09, Pascal, C, and Cobol. In Basic09 and Pascal, "GET", "PUT", and "SEEK" functions operate on random-access files.

The file is organized as an ordered sequence of "records". Each record has exactly the same length, so given a record's numerical index, the record's beginning address within the file can be computed by multiplying the record number by the number of bytes used for each record. Thus, records can be directly accessed in any order.

In most cases, the high-level language allows each record to be subdivided into "fields". Each field generally has a fixed length and usage for all records within the file. For example, the first field of a record may be defined as being 25 text characters, the next field may be two bytes long and used to hold 16-bit binary numbers, etc.

It is important to understand that NitrOS-9 itself does not directly process or deal with records other than providing the basic file functions required by all high-level languages to create and use random-access files.

3.10.4. Executable Program Module Files

These files are used to hold program modules generated by the assembler or *compiled* by high-level languages. Each file may contain *one or more* program modules.

NitrOS-9 program modules resident in memory have a standard module format that, besides the object code, includes a "module header" and a CRC check value. Program module(s) stored in files contain exact binary copies of the programs as they will exist in memory, and not one byte more. NitrOS-9 does not require a "load record" system commonly used by other operating systems because NitrOS-9 programs are position-independent code and therefore do not have to be loaded into specific memory addresses.

In order for NitrOS-9 to load the program module(s) from a file, the file itself must have execute permission and each module must have a valid module header and

CRC check value. If a program module has been altered in any way, either as a file or in memory, its CRC check value will be incorrect And NitrOS-9 will refuse to load the module. The **verify** command can be used to check the correctness of the check values, and update them to corrected values if necessary.

On Level One systems, if a file has two or more modules, they are treated as independent entities after loading and reside at different memory regions.

Like other files that contain "binary" data, attempts to "list" program files will result in the display of random characters on the terminal giving strange effects. The **dump** command can be used to safely examine the contents of this kind of file in hexadecimal and controlled ASCII format.

3.10.5. Directory Files

Directory files play a key role in the NitrOS-9 file system. They can only be created by the **makdir** command, and can be identified by the "d" attribute being set (see Section 3.9.1). The file is organized into 32-byte records. Each record can be a directory entry. The first 29 bytes of the record is a string of characters which is the file name. The last character of the name has its sign bit (most significant bit) set. If the record is not in use the first character position will have the value zero. The last three bytes of the record is a 24-bit binary number which is the logical sector number where the file header record (see Section 3.11) is located.

The **makdir** command initializes all records in a new directory to be unused entries except for the first two entries. These entries have the names . and .. along with the logical sector numbers of the directory and its parent directory, respectively (see Section 3.8.3).

Directories cannot be copied or listed - the **dir** command is used instead. Directories also cannot be deleted directly (see Section 3.6).

3.10.6. Miscellaneous File Usage

NitrOS-9's basic file functions are so versatile it is possible to devise an almost unlimited number of special-purpose file formats for particular applications, which do not fit into any of the three previously discussed categories.

Examples of this category are COBOL Indexed Sequential (ISAM) files and some special word processor file formats which allow random access of text lines. As discussed in Sec. 3.9.1, most NitrOS-9 utility commands work with any file format including these special types. In general, the **dump** command is the preferred method for examining the contents of unusually formatted files.

3.11. Physical File Organization

NitrOS-9's file system implements a universal logical organization for all I/O devices that effectively eliminates most hardware-related considerations for most applications. This section gives basic information about the physical file structure used by NitrOS-9. For more information, see the NitrOS-9 System Programmer's Manual.

Each NitrOS-9 file is comprised of one or more sectors which are the physical storage units of the disk systems. Each sector holds exactly 256 data bytes, and disk is numbered sequentially starting with sector zero, track zero. This number is called a "logical sector number", or *LSN*. The mapping of logical sector numbers to physical track/sector numbers is done by the disk driver module.

Sectors are the smallest allocatable physical unit on a disk system, however, to increase efficiency on some larger-capacity disk. systems, NitrOS-9 uses uniform-sized

groups of sectors, called *clusters*, as the smallest allocatable unit. Cluster sizes are always an integral power of two (2, 4, 8, etc.). One sector of each disk is used as a *bitmap* (usually LSN 1), in which each data bit corresponds to one cluster on the disk. The bits are set and cleared to indicate which clusters are in use (or defective), and which are free for allocation to files.

The Color Computer disk system uses the following format:

- double density recording on two sides
- 40 tracks per disk
- 18 sectors per track
- one sector per cluster

Each file has a directory entry (see Section 3.10.5) which includes the file name and the logical sector number of the file's "file descriptor sector", which contains a complete description of the file including:

- attributes
- owner
- date and time created
- size
- segment list (description of data sector blocks)

Unless the file size is zero, the file will have one or more sectors/clusters used to store data. The data sectors are grouped into one or more contiguous blocks called "segments".

Chapter 4. Advanced Features of the Shell

The basic shell functions were introduced in a prior chapter in order to provide an understanding of how basic NitrOS-9 commands work. In this section the more advanced capabilities of the shell are discussed. In addition to basic command line processing, the shell has functions that facilitate:

- I/O redirection (including filters)
- Memory Allocation
- Multitasking (concurrent execution)
- Procedure File Execution (background processing)
- Execution Control (built-in commands)

There is a virtually unlimited combination of ways these capabilities can be used, and it is impossible to give more than a representative set of examples in this manual. You are therefore encouraged to study the basic rules, use your imagination, and explore the possibilities on your own.

4.1. A More Detailed Description Command Line Processing

The shell is a program that reads and processes command lines one at a time from its input path (usually your keyboard). Each line is first scanned (or "parsed") in order to identify and process any of the following parts which may be present:

- A program, procedure file, or built-in command name ("verbs")
- Parameters to be passed to the program
- Execution modifiers to be processed by the shell

Note that only the verb (the program or command name) need be present, the other parts are optional. After the verb has been identified, the shell processes modifiers (if any). Any other text not yet processed is assumed to be parameters and passed to the program called.

Unless the verb is a "built-in command", the shell will run the program named as a new process (task). It then deactivates itself until the program called eventually terminates, at which time it gets another input line, then the process is repeated. This happens over and over until an end-of-file condition is detected on the shell's input path which causes the shell to terminate its own execution.

Here is a sample shell line which calls the assembler:

```
asm sourcefile l -o >/p #12k
```

In this example:

asm	is the verb
sourcefile l -o	are parameters passed to **asm**
>/p	is a modifier which redirects the output (listing) to the system's printer
#12K	is a modifier which requests that the process be assigned 12K bytes of memory instead of its (smaller) default amount.

The verb must be the first name in the command line. After it has been scanned, the shell first checks if it is a "built-in" command. If it is, it is immediately executed. Otherwise, the shell assumes it is a program name and attempts to locate and execute

it.

4.2. Execution Modifiers

Execution modifiers are processed by the shell before the program is run. If an error is detected in any of the modifiers, the run will be aborted and the error reported. Characters which comprise modifiers are stripped from the part(s) of the command line passed to the program as parameters, therefore, the characters reserved for use as modifiers (# ; ! < > &) cannot be used inside parameters, but can be used before or after the parameters.

4.2.1. Alternate Memory Size Modifier

When command programs are invoked by the shell, they are allocated the minimum amount of working RAM memory specified in the program's module header. A module header is part of all executable programs and holds the program's name, size, memory requirements, etc. Sometimes it is desirable to increase this default memory size. Memory can be assigned in 256-byte pages using the modifier "#n" where n is the decimal number of pages, or in 1024 byte increments using the modifier "#nK". The two examples below behave identically:

```
OS9: copy #8 file1 file2     (gives 8*256 = 2048 bytes)
OS9: copy #2K file1 file2    (gives 2*1024 = 2048 bytes)
```

4.2.2. I/O Redirection Modifiers

The second kind of modifier is used to redirect the program's "standard I/O paths" to alternate files or devices. Well-written NitrOS-9 programs use these paths for routine I/O. Because the programs do not use specific file or device names, it is fairly simple to "redirect" the I/O to any file or device without altering the program itself. Programs which normally receive input from a terminal or send output to a terminal use one or more of the standard I/O paths as defined below:

STANDARD INPUT: This path normally passes data from the terminal's keyboard to the program.

STANDARD OUTPUT PATH: This path is normally used to output data from the program to the terminal's display.

STANDARD ERROR OUTPUT PATH: This path is used to output routine status messages such as prompts and errors to the terminal's display (defaults to the same device as the standard output path). NOTE: The name "error output" is sometimes misleading since many other kinds of messages besides errors are sent on this path.

When new processes are created, they inherit their parent process' standard I/O paths. Therefore, when the shell creates new processes, they usually inherit its standard I/O paths. When you log-on the shell's standard input is the terminal keyboard; the standard output and error output is the terminal's display. When a redirection modifier is used on a shell command line, the shell will open the corresponding paths and pass them to the new process as its standard I/O paths. There are three redirection modifiers as given below:

<	Redirect the standard input path
>	Redirect the standard output path
>>	Redirect the standard error output path

When redirection modifiers are used on a command line, they must be immediately

followed by a pathlist describing the file or device the I/O is to be redirected to or from. For example, the standard output of **list** can be redirected to write to the system printer instead of the terminal:

```
OS9: list correspondence >/p
```

Files referenced by I/O redirection modifiers are automatically opened or created, and closed (as appropriate) by the shell. Here is another example, the output of the **dir** command is redirected to the file `/D1/savelisting`:

```
OS9: DIR >/D1/savelisting
```

If the **list** command is used on the file `/D1/savelisting`, output from the **dir** command will be displayed as shown below:

```
OS9: list /d1/savelisting

   Directory of .   10:15:00
myfile          savelisting       file1
```

Redirection modifiers can be used before and/or after the program's parameters, but each modifier can only be used once.

4.3. Command Separators

A single shell input line can request execution of more than one program. These programs may be executed sequentially or concurrently. Sequential execution means that one program must complete its function and terminate before the next program is allowed to begin execution. Concurrent execution means that several programs are allowed to begin execution and run simultaneously.

4.3.1. Sequential Execution

Programs are executed sequentially when each is entered on a separate line. More than one program can be specified on a single shell command line by separating each *program name parameters* from the next one with a ";" character. For example:

```
OS9: copy myfile /d1/newfile ; dir >/p
```

This command line will first execute the **copy** command and then the **dir** command.

If an error is returned by any program, subsequent commands on the same line are not executed (regardless of the state of the "x" option), otherwise, ";" and "return" are identical separators.

Here are some more examples:

```
OS9: copy oldfile newfile; del oldfile; list newfile

OS9: dir >/d1/myfile ; list temp >/p; del temp
```

All programs executed sequentially are in fact separate, child processes of the shell. After initiating execution of a program to be executed sequentially, the shell enters the "wait" state until execution of the called program terminates.

4.3.2. Concurrent Execution

The second kind of separator is the "&" which implies concurrent execution, meaning that the program is run (as a separate, child process), but the shell does not wait for it to complete before processing the next command.

The concurrent execution separator is therefore the means by which multiprogramming (running two or more programs simultaneously) is accomplished. The number of programs that can run at the same time is not fixed: it depends upon the amount of free memory in the system versus the memory requirements of the specific programs. Here is an example:

```
OS9: dir >/p&
&007

OS9:
```

This command line will cause shell to start the **dir** command executing, print the process ID number (&007), and then immediately display the "OS9:" prompt and wait for another command to be entered. Meanwhile the **dir** command will be busy sending a directory listing to the printer. You can display a "status summary" of all processes you have created by using the **procs** command. Below is another example:

```
OS9: dir >/p& list file1& copy file1 file2 ; del temp
```

Because they were followed by "&" separators, the **dir**, **list**, and **copy** programs will run concurrently, but the **del** program will not run until the **copy** program has terminated because sequential execution (";") was specified.

4.3.3. Pipes and Filters

The third kind of separator is the "!" character which is used to construct "pipelines". Pipelines consist of two or more concurrent programs whose standard input and/or output paths connect to each other using "pipes".

Pipes are the primary means-by which data is transferred from process to process (interprocess communications). Pipes are first-in, first-out buffers that behave like mass-storage files.

I/O transfers using pipes are automatically buffered and synchronized. A single pipe may have several "readers" and several "writers". Multiple writers send, and multiple readers accept, data to/from the pipe on a first-come, first-serve basis. An end-of-file will occur if an attempt is made to read from a pipe but there are no writers available to send data. Conversely, a write error will occur if an attempt is made to write to a pipe having no readers.

Pipelines are created by the shell when an input line having one or more "!" separators is processed. For each "!", the standard output of the program named to the left of the "!" is redirected via a pipe to the standard input of the program named to the right of the "!". Individual pipes are created for each "!" present. For example:

```
OS9: update <master_file ! sort ! write_report >/p
```

In the example above, the program **update** has its input redirected from a path called `master_file`. Its standard output becomes the standard input for the program **sort**. Its output, in turn, becomes the standard input for the program **write_report**, which has its standard output redirected to the printer.

All programs in a pipeline are executed concurrently. The pipes automatically synchronize the programs so the output of one never "gets ahead" of the input request of the next program in the pipeline. This implies that data cannot flow through a pipeline any faster than the slowest program can process it. Some of the most useful

applications of pipelines are jobs like character set conversion, print file formatting, data compression/decompression, etc. Programs which are designed to process data as components of a pipeline are often called "filters". The **tee** command, which uses pipes to allow data to be simultaneously "broadcast" from a single input path to several output paths, is a useful filter.

4.4. Command Grouping

Sections of shell input lines can be enclosed in parentheses which permits modifiers and separators to be applied to an entire set of programs. The shell processes them by calling itself recursively (as a new process) to execute the enclosed program list. For example:

```
OS9: (dir /d0; dir /d1) >/p
```

gives the same result as:

```
OS9: dir /d0 >/p; dir /d1 >/p
```

except for the subtle difference that the printer is "kept" continuously in the first example; in the second case another user could "steal" the printer in between the **dir** commands.

Command grouping can be used to cause a group of programs to be executed sequentially, but also concurrently with respect to the shell that initiated them, such as:

```
OS9: (del file1; del file2; del file3)&
```

A useful extension of this form is to construct pipelines consisting of sequential and/or concurrent programs. For example:

```
OS9: (dir CMDS; dir SYS) ! makeuppercase ! transmit
```

Here is a very practical example of the use of pipelines. Recall that the **dsave** command generates a procedure file to copy all the files in a directory. The example below shows how the output of **dsave** can be pipelined to a shell which will execute the NitrOS-9 commands as they are generated by **dsave**. Assume that we want to copy all files from a directory called `WORKING` to a directory called `ARCHIVE`:

```
OS9: chd /d0/WORKING; dsave /d0/ARCHIVE ! shell -p
```

4.5. Built-in Shell Commands and Options

When processing input lines, the shell looks for several special names of commands or option switches that are built-in the shell. These commands are executed without loading a program and creating a new process, and generally affect how the shell operates. They can be used at the beginning of a line, or following any program separator (";", "&", or "!"). Two or more adjacent built-in commands can be separated by spaces or commas.

The built-in commands and their functions are:

chd *pathlist*	change the working data directory to the directory specified by the pathlist.
chx *pathlist*	change the working execution directory to the directory specified by the pathlist.

ex *name*	directly execute the module named. This transforms the shell process so it ceases to exist and a new module begins execution in its place.
w	wait for any process to terminate.
* *text*	comment: "text" is not processed.
kill *Proc ID*	abort the process specified.
setpr *Proc ID priority*	changes process' priority.
x	causes shell to abort on any error (default)
-x	causes shell not to abort on error
p	turns shell prompt and messages on (default)
-p	inhibits shell prompt and messages
t	makes shell copy all input lines to output
-t	does not copy input lines to output (default)

The change directory commands switch the shell's working directory and, by inheritance, any subsequently created child process. The **ex** command is used where the shell is needed to initiate execution of a program without the overhead of a suspended **shell** process. The name used is processed according to standard shell operation, and modifiers can be used.

4.6. Shell Procedure Files

The shell is a reentrant program that can be simultaneously executed by more than one process at a time. As is the case with most other NitrOS-9 programs, it uses standard I/O paths for routine input and output. specifically, it requests command lines from the standard input path and writes its prompts and other data to the standard error path.

The shell can start up another process also running the shell by means of the **shell** command. If the standard input path is redirected to a mass storage file, the new "incarnation" of the shell can accept and execute command lines from the file instead of a terminal keyboard. The text file to be processed is called a "procedure file". It contains one or more command lines that are identical to command lines that are manually entered from the keyboard. This technique is sometimes called "batch" or "background" processing.

If the *program name* specified on a shell command line can not be found in memory or in the execution directory, shell will search the data directory for a file with the desired name. If one is found, shell will automatically execute it as a procedure file.

Execution of procedure files have a number of valuable applications. It can eliminate repetitive manual entry of commonly-used sequences of commands. It can allow the computer to execute a lengthy series of programs "in the background" while the computer is unattended or while the user is running other programs "in the foreground".

In addition to redirecting the shell's standard input to a procedure file, the standard output and standard error output can be redirected to another file which can record output for later review or printing. This can also eliminate the sometimes-annoying output of shell messages to your terminal at random times.

Here are two simple ways to use the shell to create another shell:

```
OS9: shell <procfile

OS9: procfile
```

Both do exactly the same thing: execute the commands of the file `procfile`. To run the procedure file in a "background" mode you simply add the ampersand operator:

```
OS9: procfile&
```

NitrOS-9 does not have any constraints on the number of jobs that can be simultaneously executed as long as there is memory available. Also, the procedure files can themselves cause sequential or concurrent execution of additional procedure files. Here's a more complex example of initiating two processing streams with redirection of each shell's output to files:

```
OS9: proc1 T >>stat1& proc2 T >>stat2&
```

Note that the built-in command "T" (copy input lines to error output) was used above. They make the output file contain a record of all lines executed, but without useless "OS9" prompts intermixed. The "-x" built-in command can be used if you do *not* want processing to stop if an error occurs. Note that the built-in commands only affect the shell that executes them, and not any others that may exist.

4.7. Error Reporting

Many programs (including the shell) use NitrOS-9's standard error reporting function, which displays an error number on the error output path. The standard error codes are listed in the Appendix A of this manual. If desired, the **printerr** command can be executed, which replaces the smaller, built-in error display routine with a larger (and slower) routine that looks up descriptive error messages from a text file called `/dd/sys/errmsg`. Once the **printerr** command has been run it cannot be turned off. Also, its effect is system-wide.

Programs called by the shell can return an error code in the CPU's "B" register (otherwise B should be cleared) upon termination. This type of error, as well as errors detected by the shell itself, will cause an error message to be displayed and processing of the command line or procedure file to be terminated unless the "-x" built-in command has been previously executed.

4.8. Running Compiled Intermediate Code Programs

Before the shell executes a program, it checks the program module's language type. If its type is not 6809 machine language, shell will call the appropriate run-time system for that module. Versions of the shell supplied for various systems are capable of calling different run-time systems. Most versions of shell call Basic09 when appropriate, and Level Two versions of shell can also call the Pascal P-code interpreter (PascalN), or the CIS Cobol runtime system (RunC).

For example, if you wanted to run a Basic09 I-code module called **adventure**, you could type the command given below:

```
OS9: basic09 adventure
```

Or you could accomplish the same thing by typing the following:

```
OS9: adventure
```

4.9. Setting Up Timesharing System Procedure Files

NitrOS-9 systems used for timesharing usually have a procedure file that brings the system up by means of one simple command or by using the system `startup` file. A procedure file which initiates the timesharing monitor for each terminal is executed

to start up the system. The procedure file first starts the system clock, then initiates concurrent execution of a number of processes that have their I/O redirected to each timesharing terminal.

Usually one **tsmon** command program is started up concurrently for each terminal in the system. This is a special program which monitors a terminal for activity. When a carriage return character is typed on any of these terminals, the **tsmon** command initiates the **login** command program. If a user does not enter a correct password or user number in three tries, the **login** command will be aborted. Here's a sample procedure file for a 4-terminal timesharing system having terminals names "TERM", "T1", "T2", and "T3".

```
* system startup procedure file
echo Please Enter the Date and Time
setime </term
printerr
tsmon /t1&
tsmon /t2&
tsmon /t3&
```

NOTE: This **login** procedure will not work until a password file called `/DD/SYS/PASSWORD` has been created. For more information, please see the **login** command description.

The example above deserves special attention. Note that the **setime** command has its input redirected to the system console "term", which is necessary because it would otherwise attempt to read the time information from its current standard input path, which is the procedure file and not the keyboard.

Chapter 5. Multiprogramming and Memory Management

One of NitrOS-9's most extraordinary abilities is multiprogramming, which is sometimes called timesharing or multitasking. Simply states, NitrOS-9 lets you computer run more than one program at the same time. This can be a tremendous advantage in many situations. For example, you can be editing one program while another is being printed. Or you can use your Color Computer to control household automation and still be able to use it for routine work and entertainment.

NitrOS-9 uses this capability all the time for internal functions. The simple way for you to do so is by putting a "&" character at the end of a command line which causes the shell to run your command as a "background task".

The information presented in this chapter is intended to give you an insight into how NitrOS-9 performs this amazing feat. You certainly don't have to know every detail of how multiprogramming works in order to use NitrOS-9, but a basic working knowledge can help you discover many new ways to use your Color Computer.

In order to allow several programs to run simultaneously and without interference, NitrOS-9 must perform many coordination and resource allocation functions. The major system resources managed by NitrOS-9 are:

CPU Time
Memory
The input/output system

In order for the computer to have reasonable performance, these resources must be managed in the most efficient manner possible. Therefore, NitrOS-9 uses many techniques and strategies to optimize system throughput and capacity.

5.1. Processor Time Allocation and Timeslicing

CPU time is a resource that must be allocated wisely to maximize the computer's throughput. It is characteristic of many programs to spend much unproductive time waiting for various events, such as an input/output operation. A good example is an interactive program which communicates with a person at a terminal on a line-by line basis. Every time the program has to wait for a line of characters to be typed or displayed, it (typically) cannot do any useful processing and would waste CPU time. An efficient multiprogramming operating system such as NitrOS-9 automatically assigns CPU time to only those programs that can effectively use the, time.

NitrOS-9 uses a technique called *timeslicing* which allows processes to share CPU time with all other active processes. Timeslicing is implemented using both hardware and software functions. The system's CPU is interrupted by a real time clock many (60 in the Color Computer) times each second. This basic time interval is called a "tick", hence, the interval between ticks is a time slice. This technique is called timeslicing because each second of CPU time is sliced up to be shared among several processes. This happens so rapidly that to a human observer all processes appear to execute continuously, unless the computer becomes overloaded with processing. If this happens, a noticeable delay in response to terminal input may occur, or "batch" programs may take much longer to run than they ordinarily do. At any occurrence of a tick, NitrOS-9 can suspend execution of one program and begin execution of another. The starting and stopping of programs is done in a manner that does not affect the program's execution. How frequently a process is given time slices depends upon its assigned priority relative to the assigned priority of other active processes.

The percentage of CPU time assigned to any particular process cannot be exactly computed because there are dynamic variables such as time the process spends waiting for I/O devices. It can be roughly approximated by dividing the process's priority by the sum of the priority numbers of all processes:

```
                        Process Priority
Process CPU Share = -------------------
                         Sum of All Active
                        Process' Priorities
```

5.2. Process States

The CPU time allocation system automatically assigns programs one of three "states" that describe their current status. Process states are also important for coordinating process execution. A process may be in one and only one state at any instant, although state changes may be frequent. The states are:

ACTIVE: processes which can currently perform useful processing. These are the only processes assigned CPU time.

WAITING: processes which have been suspended until another process terminates. This state is used to coordinate execution of sequential programs. The shell, for example, will be in the waiting state during the time a command program it has initiated is running.

SLEEPING: processes suspended by self-request for a specified time interval or until receipt of a "signal". Signals are internal messages used to coordinate concurrent processes. This is the typical state of programs which are waiting for input/output operations.

Sleeping and waiting processes are not given CPU time until they change to the active state.

5.3. Creation of New Processes

The sequence of operations required to create a new process and initially allocate its resources (especially memory) are automatically performed by NitrOS-9's "fork" function. If for any reason any part of the sequence cannot be performed the fork is aborted and the prospective parent is passed an appropriate error code. The most frequent reason for failure is unavailablity of required resources (especially memory) or when the program specified to be run cannot be found. A process can create many new processes, subject only to the limitation of the amount of unassigned memory available.

When a process creates a new process, the creator is called the "parent process", and the newly created process is called the "child process". The new child can itself become a parent by creating yet another process. If a parent process creates more than one child process, the children are called "siblings" with respect to each other. If the parent/child relationship of all processes in the system is examined, a hierarchical lineage becomes evident. In fact, this hierarchy is a tree structure that resembles a family tree. The "family" concept makes it easy to describe relationships between processes, and so it is used extensively in descriptions of NitrOS-9's multiprogramming operations.

When the parent issues a fork request to NitrOS-9, it must specify the following required information:

- A PRIMARY MODULE, which is the name of the program to be executed by the new process. The program can already be present in memory, or NitrOS-9 may load it from a mass storage file having the same name.
- PARAMETERS, which is data specified by the parent to be passed to and used by the new process. This data is copied to part of the child process' memory area. Parameters are frequently used to pass file names, initialization values, etc. The shell, passes command line parameters this way.

The new process also "inherits" copies of certain of its parent's properties. These are:

- A USER NUMBER which is used by the file security system and is used to identify all processes belonging to a specific user (this is not the same as the "process ID", which identifies a specific process) . This number is usually obtained from the system password file when a user logs on. The system manager always is user number zero.
- STANDARD INPUT AND OUTPUT PATHS: the three paths (input, output, and error/status) used for routine input and output. Note that most paths (files) may be shared simultaneously by two or more processes. The two current working directories are also inherited.
- PROCESS PRIORITY which determines what proportion of CPU time the process receives with respect to others.

As part of the fork operation, NitrOS-9 automatically assigns:

- A PROCESS ID: a number from 1 to 255, which is used to identify specific processes. Each process has a unique process ID number.
- MEMORY: enough memory required for the new process to run. Level Two systems give each process a unique "address space". In Level One systems, all processes share the single address space. A "data area", used for the program's parameters, variables, and stack is allocated for the process' exclusive use. A second memory area may also be required to load the program (primary module) if it is not resident in memory.

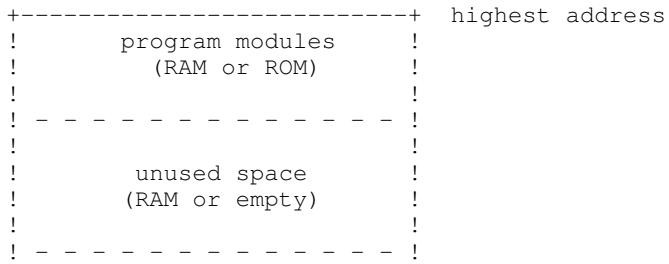
To summarize, the following items are given to or associated with new processes:

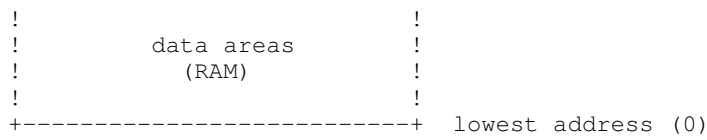
- Primary Module (program module to be run)
- Parameter(s) passed from parent to child
- User Number
- Standard I/O paths and working directories
- Process Priority
- Process ID
- Memory

5.4. Basic Memory Management Functions

An important NitrOS-9 function is memory management. NitrOS-9 automatically allocates all system memory to itself and to processes, and also keeps track of the logical *contents* of memory (meaning which program modules are resident in memory at any given time). The result is that you seldom have to be bothered with the actual memory addresses of programs or data.

Within the address space, memory is assigned from higher addresses downward for program modules, and from lower addresses upward for data areas, as shown below:





5.4.1. Loading Program Modules Into Memory

When performing a fork operation, NitrOS-9's first step is to attempt to locate the requested program module by searching the "module directory", which has the address of every module present in memory. The 6809 instruction set supports a type of program called "reentrant code" which means the exact same "copy" of a program can be shared by two or more different processes simultaneously without affecting each other, provided that each "incarnation" of the program has am independent memory area for its variables.

Almost all NitrOS-9 family software is reentrant and can make most efficient use of memory. For example, Basic09 requires 22K bytes of memory to load into. If a request to run Basic09 is made, but another user (process) had previously caused it to be loaded into memory, both processes will share the same copy, instead of causing another copy to be loaded (which would use an additional 22K of memory). NitrOS-9 automatically keeps track of how many processes are using each program module and deletes the module (freeing its memory for other uses) when all processes using the module have terminated.

If the requested program module is not already in memory, the name is used as a pathlist (file name) and an attempt is made to load the program from mass storage.

Every program module has a "module header" that describes the program and its memory requirements. NitrOS-9 uses this to determine how much memory for variable storage should be allocated to the process (it can be given more memory by specifying an optional parameter on the shell command line). The module header also includes other important descriptive information about the program, and is an essential part of NitrOS-9 operation at the machine language level. A detailed description of memory modules and module headers can be found in the "NitrOS-9 System Programmer's Manual".

Programs can also be explicitly loaded into memory using the **load** command. As with fork, the program will actually be loaded only if it is not already in memory. If the module is not in memory, NitrOS-9 will copy a candidate memory module from the file into memory, verify the CRC, and then, if the module is not already in the module directory, add the module to the directory. This process is repeated until all the modules in the file are loaded, the 64K memory limit is exceeded, or until a module with an invalid format is encountered. NitrOS-9 always links to the first module read from the file.

If the program module *is* already in memory, the load will proceed as described above, loading the module from the specified file, verifying the CRC, and when attempting to add the valid module to the module directory, noticing that the module is already known, the load merely increments the known module's link count (the number of processes using the module.) The load command can be used to "lock a program into memory. This can be useful if the same program is to be used frequently because the program will be kept in memory continuously, instead of being loaded repeatedly.

The opposite of **load** is the **unlink** command, which decreases a program module's link count by one. Recall that when this count becomes zero (indicating the module in no longer used by any process), the module is deleted, e.g., its memory is deallocated and its name is removed from the module directory. The **unlink** command is generally used in conjunction with the **load** command (programs loaded by fork are automatically unlinked when the program terminates).

Here is an example of the use of **load** and **unlink** to lock a program in memory. Suppose the **copy** command will be used five times. Normally, the copy command would be loaded each time the **copy** command is called. If the **load** command is used first, **copy** will be locked into memory first, for example:

```
OS9: load copy
OS9: copy file1 file1a
OS9: copy file2 file2a
OS9: copy file3 file3a
OS9: unlink copy
```

It is important to use the **unlink** command after the program is no longer needed, or the program will continue to occupy memory which otherwise could be used for other purposes. Be very careful *not* to completely unlink modules in use by any process! This will cause the memory used by the module to be deallocated and its contents destroyed. This will certainly cause all programs using the unlinked module to crash.

5.4.2. Loading Multiple Programs

Another important aspect of program loading is the ability to have two or more programs resident in memory at the same time. This is possible because all NitrOS-9 program modules are "position-independent code", or "PIC". PIC programs do not have to be loaded into specific, predetermined memory addresses to work correctly, and can therefore be loaded at different memory addresses at different times. PIC programs require special types of machine language instructions which few computers have. The ability of the 6809 microprocessor to use this type of program is one of its most powerful features.

The **load** command can therefore be used two or more times (or a single file may contain several memory modules), and each program module will be automatically loaded at different, non-overlapping addresses (most other operating systems write over the previous program's memory whenever a new program is loaded). This technique also relieves the user from having to be directly concerned with absolute memory addresses. Any number of program modules can be loaded until available system memory is full.

5.4.3. Memory Fragmentation

Even though PIC programs can be initially loaded at any address where free memory is available, program modules cannot be relocated dynamically afterwards, e.g., once a program is loaded it must remain at the address at which it was originally loaded (however Level Two systems can "load" (map) memory resident programs at different addresses in each process' address space). This characteristic can lead to a sometimes troublesome phenomenon called "memory fragmentation". When programs are loaded, they are assigned the first sufficiently large block of memory at the highest address possible in the address space. If a number of program modules are loaded, and subsequently one or more modules which are located in between other modules are "unlinked", several fragments of free memory space will exist. The sum of the sizes of the free memory space may be quite large, but because they are scattered, not enough space will exist in a single block to load a program module larger than the largest free space.

The **mfree** command shows the location and size of each unused memory area and the **mdir -e** command shows the address, size, and link (use) count of each module in the address space. These commands can be used to detect fragmentation. Memory can usually be de-fragmemted by unlinking scattered modules and reloading them. *Make certain* none are in use before doing so.

Chapter 6. Use of the System Disk

Disk-based NitrOS-9 systems use a system disk to load many parts of the operating system during the system startup and to provide files frequently used during normal system operations. Therefore, the system disk is generally kept in disk drive zero ("/D0") when the system is running.

Two files used during the system startup operation, `OS9Boot` and `startup` *must* reside in the system disk's root directory. Other files are organized into three directories: `CMDS` (commands), `DEFS` (system-wide definitions), and `SYS` (other system files). Other files and directories created by the system manager and/or users may also reside on the system disk. These frequently include each user s initial data directory.

6.1. The OS9Boot File

The file called `OS9Boot` loaded into RAM memory by the "bootstrap" routine located in the NitrOS-9 firmware. It includes file managers, device drivers and descriptors, and any other modules which are permanently resident in memory. The NitrOS-9 distribution disk's `OS9Boot` file contains the following modules:

KernelP2	NitrOS-9 Kernel, Part 2
IOMan	NitrOS-9 Input/Output Manager
Init	Initialization Data Module
RBF	Random Block (disk) File Manager
SCF	Sequential Character (terminal) File Manager
PipeMan	Pipe File Manager
Piper	Pipe Driver
Pipe	Pipe Device Descriptor
CC3IO	CoCo 3 Keyboard/Video Device Driver
WindInt	CoCo 3 Graphics Co-Module
VDGInt	CoCo 2 Compatible Graphics Co-Module
Term	Terminal Device Descriptor
CC3Disk	CoCo 3 Disk Driver
DD, D0, D1	Disk Device Descriptors
Printer	Printer Device Driver
p	Printer Device Descriptor
Clock	Real-Time Clock Module
Clock2	Second Part of Real-Time Clock Module
SysGo	System Startup Process

Users may create new bootstrap files which may include additional modules (see **OS9Gen** command). Any module loaded as part of the bootstrap cannot be unlinked and is stored in memory with a minimum of fragmentation. It may be advantageous to include in the OS9Boot file any module used constantly during normal system operation. This can be done with the OS9GEN command.

6.2. The SYS Directory

The directory `/d0/SYS` contains several important files:

password	the system password file (see **login** command)
errmsg	the error message file
helpmsg	the help database file

These files (and the `SYS` directory itself) are not absolutely required to boot NitrOS-9, they are needed if **login**, **tsmon**, or **help** will be used. Users may add other system-wide files of similar nature if desired.

6.3. The Startup File

The file `startup` in the root directory is a shell procedure file which is automatically processed immediately after system startup. The user may include in `startup` any legal shell command line. Often this will include **setime** to start the system clock. If this file is not present the system will still start correctly but the user must run the SETIME command manually.

6.4. The CMDS Directory

The directory `CMDS` is the system-wide command object code directory, which is normally shared by all users as their working execution directory. If **shell** is not part of the `OS9Boot` file (and it shouldn't be in a Level 2 system), it must be present in this directory. The system startup process "sysgo" makes `CMDS` the initial execution directory.

6.5. The DEFS Directory

The directory `DEFS` is a directory that contains assembly language source code files which contain common system-wide symbolic definitions, and are normally included in assembly language programs by means of the NitrOS-9 Assembler "use" directive. The presence and use of this directory is optional, but highly recommended for any system used for assembly language programs. The files commonly contained in this directory are:

OS9Defs	main system-wide definition file
RBFDefs	RBF file manager definition file
SCFDefs	SCF file manager definition file
Systype	System types definition file

6.6. Changing System Disks

The system disk is not usually removed while the system is running, especially on multiuser systems. If it is, the **chx** and **chd** (if the working data directory was on the system disk) commands should be executed to reset the working directory pointers because the directories may be at different addresses on the new disk, for example:

```
chx /d0/cmds
chd /d0
```

In general, it is unwise to remove a disk and replace it with another if any paths are open to files resident on the disk. It is *dangerous* to exchange *any* disk if any files on it

are open in WRITE or UPDATE modes.

6.7. Making New System Disks

To make a system disk, the following steps must be performed:

1. The new disk must be formatted.
2. The `OS9Boot` file must be created and linked by the **OS9Gen** or **Cobbler** commands.
3. The `startup` file must be created or copied.
4. The `CMDS` and `SYS` directories and the files they contain must be copied.
5. For Level 2, the `sysgo` file in the root directory must be copied.

Steps 2 through 5 may be performed manually, or automatically by any of the following methods:

1. By a shell procedure file created by the user.
2. By a shell procedure file generated by the **dsave** command
3. By the **backup** command

Chapter 7. System Command Descriptions

This section contains descriptions for each of the command programs that are supplied with NitrOS-9. These programs are usually called using the shell, but can be called from most other NitrOS-9 family programs such as BASIC09, Interactive Debugger, Macro Text Editor, etc. Unless otherwise noted, these programs are designed to run as individual processes.

> **Warning**
>
> Although many NitrOS-9 commands may work on Level 1 or Level 2 systems, there are differences. Take care not to mix command files from Level 1 systems on Level 2, or the reverse.

7.1. Formal Syntax Notation

Each command description includes a syntax definition which describes how the command sentence can be constructed. These are symbolic descriptions that use the following notation:

[ ]	= Brackets indicate that the enclosed item(s) are optional.
{ }	= Braces indicate that the enclosed item(s) can be either omitted or repeated multiple times.
path	= Represents any legal pathlist.
devname	= Represents any legal device name.
nodname	= Represents any legal memory module name.
procID	= Represents a process number.
opts	= One or more options defined in the command description.
arglist	= a list of arguments (parameters).
text	= a character string terminated by end-of-line.

NOTE: The syntax of the commands given does not include the shell's built in options such as alternate memory size, I/O redirection, etc. This is because the shell will filter its options out of the command line before it is passed to the program being called.

7.2. Commands

ATTR

Name

ATTR — Change file security attributes

Synopsis

attr *path* [{ *permission abbreviations* }]

Description

This command is used to examine or change the security permissions of a file. To enter the command, type **attr** followed by the pathlist for the file who's security permissions are to be changed, followed by a list of permissions which are to be turned on or off. A permission is turned on by giving its abbreviation, or turned off by preceding its abbreviation with a minus sign. Permissions not explicitly named are not affected. If no permissions are given the current file attributes will be printed. You can not change the attributes of a file which you do not own (except for user zero, who can change the attributes of any file in the system).

The file permission abbreviations are:

d = Directory file
s = Sharable file
r = Read permit to owner
w = Write permit to owner
e = Execute permit to owner
pr = Read permit to public
pw = Write permit to public
pe = Execute permit to public

The **attr** command may be used to change a directory file to a non-directory file if all entries have been deleted from it. Since the DEL command will only delete non-directory files, this is the only way a directory may be deleted. You cannot change a non-directory file to a directory file with this command (see **makdir**).

For more information see: Section 3.9, Section 3.9.1

Examples

```
attr myfile -pr -pw

attr myfile r w e pr rw pe


attr datalog
-s-wr-wr
```

BACKUP

Name

`BACKUP` — Make a backup copy of a disk

Synopsis

backup [`e`] [`s`] [`-v`] [*`devname`* [*`devname`*]]

Description

This command is used to physically copy all data from one device to another. A physical copy is performed sector by sector without regard to file structures. In almost all cases the devices specified mun.t have the exact same format (size, density, etc.) and must not have defective sectors.

If both device name are omitted the names "/d0" and "/d1" are assumed. If the second device name is omitted, a single unit backup will be performed on the drive specified.

The options are:

E = Exit if any read error occurs.
S = Print single drive prompt message.
-V = Do not verify.
#nK = more memory makes backup run faster

Examples

```
backup /D2 /D3

backup -V



OS9: backup

Ready to BACKUP from /D0 to /D1 ?: Y
MYDISK is being scratched
OK ?: Y
Number of sectors copied: $04D0
Verify pass
Number of sectors verified: $04D0
OS9:
```

Below is an example of a single drive backup. **backup** will read a portion of the source disk into memory, you remove the source disk and place the destination disk into the drive, **backup** writes on the destination disk, you remove the destination disk and place the source disk into the drive. This continues until the entire disk has been copied. Giving **backup** as much memory as possible will cause fewer disk exchanges to be required.

For more information see: Section 1.2.2

```
OS9:backup /D0 #10k

Ready to BACKUP from /D0 to /D0 ?: Y
Ready DESTINATION, hit a key:
MYDISK is being scratched
OK ?: Y
Ready SOURCE, hit a key:
Ready DESTINATION, hit a key:
Ready SOURCE, hit a key:
Ready DESTINATION, hit a key:

(several repetitions)

Ready DESTINATION, hit a key:
Number of sectors copied: $4D0
Verify pass
Number of sectors verified: $4D0
```

BINEX

Name

BINEX — Convert Binary To S-Record File

Synopsis

binex *path1 path2*

Description

S-Record files are a type of text file that contains records that represent binary data in hexadecimal character form. This Motorola-standard format is often directly accepted by commercial PROM programmers, emulators, logic analyzers and similar devices that are interfaced RS-232 interfaces. It can also be useful for transmitting files over data links that can only handle character-type data; or to convert NitrOS-9 assembler or compiler-generated programs to load on non-NitrOS-9 systems.

Binex converts "path1", a NitrOS-9 binary format file, to a new file named "path2" in S-Record format. If invoked on a non-binary load module file, a warning message is printed and the user is asked if **binex** should proceed anyway. A "Y" response means yes; any other answer will terminate the program. S-Records have a header record to store the program name for informational purposes and each data record has an absolute memory address which is not meaningful to NitrOS-9 since it uses position-independent-code. However, the S-Record format requires them so **binex** will prompt the user for a program name and starting load address. For example:

```
binex /d0/cmds/scanner scanner.S1
Enter starting address for file: $100
Enter name for header record: scanner
```

To download the program to a device such as a PROM programmer (for example using serial port T1) type:

```
list scanner.S1 >/T1
```

BUILD

Name

BUILD — Build a text file from standard input

Synopsis

build *path*

Description

This command is used to build short text files by copying the standard input path into the file specified by *path*. **Build** creates a file according to the pathlist parameter, then displays a "?" prompt to request an input line. Each line entered is written to the output path (file). Entering a line consisting of a carriage return only causes **build** to terminate.

Example:

```
build small_file
build /p                    (copies keyboard to printer)
```

The standard input path may also be redirected to a file. Below is an example:

```
build <mytext /T2      (copies file "mytext" to terminal T2)


OS9: build newfile

? The powers of the NitrOS-9
? operating system are truly
? fantastic.
? [RETURN]

OS9: list newfile

The powers of the NitrOS-9
operating system are truly
fantastic.
```

CHD/CHX

Name

CHD/CHX — Change working data directory / Change working execution directory

Synopsis

chd *pathlist*

chx *pathlist*

Description

These are shell "built in" commands used to change NitrOS-9's working data directory or working execution directory. Many commands in NitrOS-9 work with user data such as text files, programs, etc. These commands assume that a file is located in the working data directory. Other NitrOS-9 commands will assume that a file is in the working execution directory.

NOTE: These commands do not appear in the CMDS directory as they are built-in to the **shell**.

For more information see: Section 3.8, Section 3.8.2

Examples

```
chd /d1/PROGRAMS

chx ..

chx binary_files/test_programs

chx /D0/CMDS; chd /D1
```

CMP

Name

CMP — File Comparison Utility

Synopsis

cmp *file1 file2*

Description

Opens two files and performs a comparison of the binary values of the corresponding data bytes of the files. If any differences are encountered, the file offset (address) and the values of the bytes from each file are displayed in hexadecimal.

The comparison ends when end-of-file is encountered on either file. A summary of the number of bytes compared and the number of differences found is then displayed.

Examples

```
OS9: cmp red blue

 Differences

byte      #1 #2
========  == ==
00000013  00 01
00000022  B0 B1
0000002A  9B AB
0000002B  3B 36
0000002C  6D 65

Bytes compared:   0000002D
Bytes different:  00000005
```

```
OS9: cmp red red

 Differences
   None ...

Bytes compared:   0000002D
Bytes different:  00000000
```

COBBLER

Name

COBBLER — Make a bootstrap file

Synopsis

cobbler *device name*

Description

Cobbler is used to create the OS9Boot file required on any disk from which NitrOS-9 is to be bootstrapped. The boot file will consist of the *same modules which were loaded into memory during the most recent boostrap.* To add modules to the bootstrap file use the **OS9Gen** command. **Cobbler** also writes the NitrOS-9 kernel on the eighteen sectors of track 34, and excludes these sectors from the disk allocation map. If any files are present on these sectors **cobbler** will display an error message.

NOTE: The boot file must fit into one contiguous block on the mass-storage device. For this reason **cobbler** is normally used on a freshly formatted disk. If **cobbler** is used on a disk and there is not a contiguous block of storage large enough to hold the boot file, the old boot file may have been destroyed and NitrOS-9 will not be able to boot from that disk until it is reformatted.

For more information see: Section 1.2.2, Section 6.1

Examples

```
OS9: cobbler /D1
```

COPY

Name

COPY — Copy data from one path to another

Synopsis

copy *path path* [-a -p -r -s -w=<dir> -x]

Description

This command copies data from the first file or device specified to the second. The first file or device must already exist, the second file is automatically created if the second path is a file on a mass storage device. Data may be of any type and is NOT modified in any way as it is copied.

Data is transferred using large block reads and writes until end-of-file occurs on the input path. Because block transfers are used, normal output processing of data does not occur on character-oriented devices such as terminals, printers, etc. Therefore, the **list** command is preferred over **copy** when a file consisting of text is to be sent to a terminal or printer.

The "-a" option will force **copy** to abort its operation if it receives an error during the copy of a file. If this option is not specified, copy will continue to attempt to copy any other files specified on its command line.

The "-p" option prevents copy from echoing the filenames that it is copying (used in conjunction with -w).

The "-r" option allows copy to rewrite the destination file if it matches the name of a source file that is being copied. If this option is not used, then the user will be prompted to overwrite a file of the same name.

The "-s" option causes **copy** to perform a single drive copy operation. The second pathlist must be a full pathlist if "-s appears. **Copy** will read a portion of the source disk into memory, you remove the source disk and place the destination disk into the drive, enter a "C" whereupon **copy** writes on the destination disk, this process continues until the entire file is copied.

The "-w=<dir>" option allows you to specify a destination directory where all the files will be copied to. Use this option when specifing multiple filenames on the command line.

The "-x=<dir>" will cause the files to be copied to an execution-relative directory.

Using the shell's alternate memory size modifier to give a large memory space will increase speed and reduce the number of media exchanges required for single drive copies.

Examples

```
copy file1 file2 #15k           (copies file1 to file2)

copy /d1/joe/news /D0/peter/messages

copy /d1/joe/news /d1/joe/weather -w=/D0/peter (where /D0/peter is a directory)

copy /term /p                   (copies console to printer)

copy /d0/cat /d0/animals/cat -s #32k
Ready DESTINATION, hit C to continue: c
Ready SOURCE, hit C to continue: c
Ready DESTINATION, hit C to continue:c
```

CPUTYPE

Name

CPUTYPE — Identify the CPU

Synopsis

cputype

Description

Identifies the CPU as 6809 or 6309.

Examples

DATE

Name

DATE — Display system date and time

Synopsis

date [-t]

Description

This command will display the current system date, and if the "-t" option is given, the current system time.

Examples

```
date -t

date -t >/p (Output is redirected to printer)

OS9: setime

       yyyy/mm/dd hh:mm:ss
Time ? 2003/04/15 14:19:00

OS9:date

April 15, 2003
```

```
OS9:date -t

April 15, 2003  14:20:20
```

DCHECK

Name

DCHECK — Check Disk File Structure

Synopsis

dcheck [-opts] *devnam*

Description

It is possible for sectors on a disk to be marked as being allocated but in fact are not actually associated with a file or the disk's free space. This can happen if a disk is removed from a drive while files are still open, or if a directory which still contains files is deleted (see Section 3.6). **Dcheck** is a diagnostic that can be used to detect this condition, as well as the general integrity of the directory/file linkages.

Dcheck is given as a parameter the name of the disk device to be checked. After verifying and printing some vital file structure parameters, **dcheck** follows pointers down the disk's file system tree to all directories and files on the disk. As it does so, it verifies the integrity of the file descriptor sectors, reports any discrepancies in the directory/file linkages, and builds a sector allocation map from the segment list associated with each file. If any file descriptor sectors (FDs) describe a segment with a cluster not within the file structure of the disk, a message is reported like:

```
*** Bad FD segment ($xxxxxx-$yyyyyy) for file: pathlist
```

This indicates that a segment starting at sector xxxxxx and ending at sector yyyyyy cannot really be on this disk. Because there is a good chance the entire FD is bad if any of it's segment descriptors are bad, the allocation map is *not* updated for corrupt FDs.

While building the allocation map, **dcheck** also makes sure that each disk cluster appears only once and only once in the file structure. If this condition is detected, **dcheck** will display a message like:

```
Cluster $xxxxxx was previously allocated
```

This message indicates that cluster xxxxxx has been found at least once before in the file structure. The message may be printed more than once if a cluster appears in a segment in more than one file.

The newly created allocation map is then compared to the allocation map stored on the disk, and any differences are reported in messages like:

```
Cluster $xxxxxx in allocation map but not in file structure
Cluster $xxxxxx in file structure but not in allocation map
```

The first message indicates sector number xxxxxx (hexadecimal) was found not to be part of the file system, but was marked as allocated in the disk's allocation map. In addition to the causes mentioned in the first paragraph, some sectors may have been excluded from the allocation map by the FORMAT program because they were defective or they may be the last few sectors of the disk, the sum of which was two small to comprise a cluster.

The second message indicates that the cluster starting at sector xxxxxx is part of the file structure but is *not* marked as allocated in the disk's allocation map. It is possible that this cluster may be allocated to another file later, overwriting the contents of the cluster with data from the newly allocated file. Any clusters that have been reported as "previously allocated" by **dcheck** as described above surely have this problem.

Available **dcheck** options are:

-w=*path*	pathlist to directory for work files
-p	print pathlists for questionable clusters
-m	save allocation map work files
-b	suppress listing of unused clusters
-s	display count of files and directories only
-o	print **dcheck**'s valid options

The "-s" option causes **dcheck** to display a count of files and directories only; only FDs are checked for validity. The "-b" option suppresses listing of clusters allocated but not in file structure. The "-p" option causes **dcheck** to make a second pass through the file structure printing the pathlists for any clusters that **dcheck** finds as "already allocated" or "in file structure but not in allocation map". The "-w=" option tells **dcheck** where to locate it's allocation map work file(s). The pathlist specified must be a FULL pathlist to a *directory*. The directory "/D0" is used is used if "-w" is not specified. It is recommended that this pathlist NOT be located on the disk being **dcheck**ed if the disk's file structure integrity is in doubt.

Dcheck builds its disk allocation map in a file called *pathlist*/DCHECKppO, where *pathlist* is as specified by the "-w=" option and pp is the process number in hexadecimal. Each bit in this bitmap file corresponds to a cluster of sectors on the disk. If the "-p" option appears on the command line, **dcheck** creates a second bitmap file (*pathlist*/DCHECKpp1) that has a bit set for each cluster **dcheck** finds as "previously allocated" or "in file structure but not in allocation map" while building the allocation map. **Dcheck** them makes another pass through the directory structure to determine the pathlists for these questionable clusters. These bitmap work files may be saved by specifying the "-m" option on the command line.

Restrictions

For best results, **dcheck** should have exclusive access to the disk being checked. Otherwise **dcheck** may be fooled if the disk allocation map changes while it is building its bitmap file from the changing file structure. **Dcheck** cannot process disks with a directory depth greater than 39 levels.

For more information see: Section 3.11, Section 3.6, **format**, 6.1 of NitrOS-9 Systems Programmer's Manual

Examples

```
OS9: dcheck /d2   (workfile is on /D0)

Volume - 'My system disk' on device /d2
```

```
$009A bytes in allocation map
1 sector per cluster
$0004D0 total sectors on media
Sector $000002 is start of root directory FD
$0010 sectors used for id, allocation map and root directory
Building allocation map work file...
Checking allocation map file...


'My system disk' file structure is intact
1 directory
2 files

OS9: dcheck -mpw=/d2 /d0
Volume - 'System disk' on device /d0
$0046 bytes in allocation map
1 sector per cluster
$00022A total sectors on media
Sector $000002 is start of root directory FD
$0010 sectors used for id, allocation map and root directory
Building allocation map work file...
Cluster $00040 was previously allocated
*** Bad FD segment ($111111-$23A6F0) for file: /d0/test/junky.file
Checking allocation map file...
Cluster $000038 in file structure but not in allocation map
Cluster $00003B in file structure but not in allocation map
Cluster $0001B9 in allocation map but not in file structure
Cluster $0001BB in allocation map but not in file structure

Pathlists for questionable clusters:
Cluster $000038 in path: /d0/OS9boot
Cluster $00003B in path: /d0/OS9boot
Cluster $000040 in path: /d0/OS9boot
Cluster $000040 in path: /d0/test/double.file

1 previously allocated clusters found
2 clusters in file structure but not in allocation map
2 clusters in allocation map but not in file structure
1 bad file descriptor sector

'System disk' file structure is not intact
5 directories
25 files
```

DEBUG

Name

DEBUG — Interactive Debugger

Synopsis

debug

Description

Interactive Debugger.

Command Summary

[SPACEBAR]expression	Evaluate; display in hexadecimal and decimal form
.	Display dot address and contents
..	Restore last dot address; display address and contents
.expression	set dot to result of expression; display address and contents
=expression	Set memory at dot to result of expression
-	Decrement dot; display address and contents
[ENTER]	Increment dot; display address and contents
:	Display all registers' contents
:register	Display the specified register's contents
:register expression	Set register to the result of expression
E module-name	Prepare for execution
G	Go to the program
G expression	Goto the program at the address specified by the result of expression
L module-name	Link to the module named; display address
B	Display all breakpoints
B expression	Set a breakpoint at the result of the expression
K	Kill all breakpoints
K expression	Kill the breakpoint at address specified by expression
M expression1 expression2	Display memory dump in tabular form
C expression1 expression2	Clear and test memory
S expression1 expression2	Search memory for pattern
$ command	Call NitrOS-9 shell with optional command
Q	Quit (exit) Debug

DED

Name

`DED` — Disk Editor

Synopsis

ded *pathlist*

Description

dEd is a screen-oriented disk editor utility. It was originally conceived as a floppy disk editor, so the display is organized around individual sectors. It performs most of the functions of Patch, from Computerware, but is faster, more compact, and screen-oriented rather than line-oriented. Individual files or the disk itself (hard, floppy, RAM) can be examined and changed, sectors can be written to an output file, and executable modules can be located, linked to and verified.

To use, type:

dEd *pathlist*

where <pathlist> is of the form: filename or dirname or /path/filename or /D0@ (edits entire disk)

dEd will read in and display the first 256 bytes in the file (disk). This is Logical Sector Number (LSN) zero. You move through the file sector (LSN) by sector using the up and down arrow keys. The current LSN number is displayed in Hex and Decimal in the upper left corner of the screen. If the disk itself was accessed (by appending '@' to it's name when **dEd** was called), the LSN is the disk sector number. If an individual file is being editted, however, the LSN displayed refers to the file, not to the disk. All numbers requested by **dEd** must be in Hex format. All commands are accessed by simply pressing the desired key.

DEL

Name

DEL — Delete a file

Synopsis

del [-x] *path* {*path*}

Description

This command is used to delete the file(s) specified by the pathlist(s). The user must have write permission for the file(s). Directory files cannot be deleted unless their type is changed to non-directory: see the **attr** command description.

If the -x option appears, the current *execution* directory is assumed.

For more information see: Section 3.6, Section 3.9.1

Examples

```
del test_program old_test_program

del /D1/number_five

OS9:dir /D1

   Directory of /D1 14:29:46
myfile          newfile

OS9:del /D1/newfile
OS9:dir /D1

   Directory of /D1 14:30:37
myfile

OS9:del myprog -x
OS9:del -x CMDS.SUBDIR/file
```

DELDIR

Name

DELDIR — Delete All Files In a Directory System

Synopsis

deldir *directory name*

Description

This command is a convenient alternative to manually deleting directories and files they contain. It is only used when *all* files in the directory system are to be deleted.

When **deldir** is run, it prints a prompt message like this:

```
OS9: deldir OLDFILES
Deleting directory file.
List directory, delete directory, or quit ? (l/d/q)
```

An "l" response will cause a **dir -e** command to be run so you can have an opportunity to see the files in the directory before they are deleted.

A "d" response will initiate the process of deleting files.

A "q" response will abort the command before action is taken.

The directory to be deleted may include directory files, which may themselves include directory files, etc. In this case, **deldir** operates recursively (e.g., it calls itself) so all lower-level directories are deleted as well. In this case the lower-level directories are processed first.

You must have correct access permission to delete all files and directories encountered. If not, **deldir** will abort upon encountering the first file for which you do not have write permission.

The **deldir** command automatically calls the DIR and ATTR commands, so they both must reside in the current execution directory.

DEVS

Name

DEVS — Show device table entries

Synopsis

devs

Description

Devs displays a list of the system's device table. The device table contains an entry for each active device known to NitrOS-9. devs does not display information for uninitialized devices. The devs display header lists the system name, the NitrOS-9 version number, and the maximum number of devices allowed in the device table.

Each line in the devs display contains five fields:

Name	Description
Device	Name of the device descriptor
Driver	Name of the device driver
File Mgr	Name of the file manager
Data Ptr	Address of the device driver's static storage
Links	Device use count

Note: Each time a user executes a chd to an RBF device, the use count of that device is incremented by one. Consequently, the Links field may be artificially high.

DMODE

Name

DMODE — Disk descriptor Editor

Synopsis

dmode [*devicename* | *-filename*] [options]

Description

This new version allows any combination of upper or lower case options to be specified.

Also, current parameters are displayed with a "$" preceding to remind the user that the values are *hexadecimal*.

Options may be prefixed with a "$". It is simply ignored.

Examples

Typical **dmode** output:

```
OS9: dmode /dd  {enter}

 drv=$00 stp=$00 typ=$80 dns=$01 cyl=$0334 sid=$06
 vfy=$00 sct=$0021 tos=$0021 ilv=$00 sas=$20
```

Now, let's say we want to change the number of cylinders this descripter shows. The following command lines would all be valid and accepted by the new **dmode**:

```
OS9:  dmode /dd CYL=276
-or-  dmode /dd Cyl=$276
-or-  dmode /dd cYL=276
```

Lastly, you may now specify either "TOS" or "T0S" to setup the number of sectors per track in track zero. Example:

```
OS9:  dmode /dd tos=21
-or-  dmode /dd t0s=21
```

DIR

Name

DIR — Display the names of files contained in a directory

Synopsis

dir [-e] [-x] [*path*]

Description

Displays a formatted list of files names in a directory file on. the standard output path. If no parameters are given, the current *data* directory is shown. If the "x" option is given, the current *execution* directory is shown. If a pathlist of a directory file is given, it is shown.

If the "e" option is included, each file's entire description is displayed: size, address, owner, permissions, date and time of last modification.

For more information see: Section 1.1.3, Section 3.5, and Section 3.9.1

Examples

```
dir                   (display data directory)

dir -x                (display execution directory)

dir -x -e             (display entire description of execution dir)

dir ..                (display parent of working data directory)

dir newstuff          (display newstuff directory)

dir -e test_programs (display entire description of test_programs)
```

DISASM

Name

`DISASM` — NitrOS-9 Module Disassembler

Synopsis

disasm [-m *`module name`* | *`filename`*] [options]

Description

Disasm was written to hack apart NitrOS-9 system modules,command modules, file managers and device drivers/descriptors either from memory or disk. Unlike most other disassemblers, **disasm** is a two pass disassembler, creating output using only referenced labels. This output can be redirected to a file and (after modifications if desired) then re-assembled.

Disasm provides completely commented disassembly of Device Descriptors... very useful for building a customized boot file.

Options

disasm -m *module name*

will link to module in memory - if not found,will load module from exec directory and then link to it...after disassembly,it will attempt to unlink the module.

disasm *pathlist/module name*

will 'read' the module from the specified path without loading.

other options:

o = display line number,address,object code & source code... useful for hard to crack modules with data embedded in the middle.

x = look for module in execution directory.

ANY combination of options is allowed (upper or lower case) but they *must* immediately follow the '-' and there must be no spaces separating the options.

DISPLAY

Name

DISPLAY — Display Converted Characters

Synopsis

display *hex* {*hex*}

Description

Display reads one or more hexadecimal numbers given as parameters, converts them to ASCII characters, and writes them to the standard output. It is commonly used to send special characters (such as cursor and screen control codes) to terminals and other I/O devices.

Examples

```
display 0C 1F 02 7F


display 15 >/p      (sends "form feed" to printer)

OS9: display 41 42 43 44 45 46
ABCDEF
```

DSAVE

Name

DSAVE — Generate procedure file to copy files

Synopsis

dsave [opts] [*path*]

Description

Dsave is used to backup or copy all files in one or more directories. It does not execute the commands; instead, it echos commands to standard output. This output can be redirected to a file and executed later as a procedure file.

When **dsave** is executed, it writes copy commands to *standard output* to copy files from the current *data* directory to the directory specified by *path*. If **dsave** encounters a directory file, it will automatically include **makdir** and **chd** commands in the output before generating copy commands for files in the subdirectory. Since **dsave** is recursive in operation, the procedure file will exactly replicate all levels of the file system from the current data directory downward (such a section of the file system is called a "subtree").

If the current working directory happens to be the root directory of the disk, **dsave** will create a procedure file that will backup the entire disk file by file. This is useful when it is necessary to copy many files from different format disks, or from floppy disk to a hard disk.

Available **dsave** options are:

-b	make output disk a system disk by using source disk's OS9Boot file,. if present.
-b=*path*	make output disk a system disk using *path* as source for the OS9Boot file.
-i	indent for directory levels
-l	do not process directories below the current level
-m	do not include **makdir** commands in procedure file
-r	forces the copy command to rewrite the file at its destination if it already exists
-s*integer*	set copy size parameter to *integer* K

For more information see: Section 2.3

Examples

Example which copies all files on "d2" to "d1":

```
chd /d0                          (select "from" directory)
dsave /d1 >/d0/makecopy       (make procedure file "makecopy")
/d0/makcopy                      (run procedure file)

chd /d0/MYFILES/STUFF
dsave -is32 /d1/BACKUP/STUFF >saver
```

```
/d0/MYFILES/STUFF/saver
```

DUMP

Name

DUMP — Formatted File Data Dump in Hexadecimal and ASCII

Synopsis

dump [*-h -m -x*] [*path*]

Description

This command produces a formatted display of the physical data contents of the path specified which may be a mass storage file or any other I/O device. If a pathlist is omitted, the standard input path is used. The output is written to standard output. This command is commonly used to examine the contents of non-text files.

The data is displayed 16 bytes per line in both hexadecimal and ASCII character format. Data bytes that have non-displayable values are represented by periods in the character area.

The addresses displayed on the dump are relative to the beginning of the file. Because memory modules are position-independent and stored on files exactly as they exist in memory, the addresses shown on the dump correspond to the relative load addresses of memory-module files.

-h	prevent dump from printing its header every 256 bytes
-m	names on the command line are modules in memory
-x	names on the command line are files relative to the execution directory

Examples

```
dump              (display keyboard input in hex)
dump myfile >/p   (dump myfile to printer)
dump -m kernel    (dump the kernel module in memory)
```

Sample Output

```
Address   0 1  2 3  4 5  6 7  8 9  A B  C D  E F   0 2 4 6 8 A C E
--------- ---- ---- ---- ---- ---- ---- ---- ----  ----------------
00000000 87CD 0038 002A P181 2800 2E00 3103 FFE0  .M.8.*q.(...1..'
00000010 0418 0000 0100 0101 0001 1808 180D 1B04  ................
00000020 0117 0311 0807 1500 002A 5445 S2CD 5343  .........*TERMSC
00000030 C641 4349 C10E 529E                      FACIA.R.
```

```
      ^                    ^                              ^

starting       data bytes in hexadecimal          data bytes in
address                format                     ASCII format
```

ECHO

Name

ECHO — Echo text to output path

Synopsis

echo *text*

Description

This command echoes its argument to the standard output path. It is typically used to generate messages in shell procedure files or to send an initialization character sequence to a terminal. The text should not include any of the punctuation characters used by the shell.

Examples

```
echo >/t2 Hello John how's it going &    (echo to t2)

echo >/term ** warning ** disk about to be scratched 1

echo >/p Listing of Transaction File; list trans >/p


OS9: echo Here is an important message!
Here is an important message!
```

EX

Name

EX — Execute program as overlay

Synopsis

ex *module name* [*modifiers*] [*parameters*]

Description

This a shell built-in command that causes the process executing the shell to start execution of another program. It permits a transition from the shell to another program without creating another process, thus conserving system memory.

This command is often used when the shell is called from another program to execute a specific program, after which the shell is not needed. For instance, applications which only use **basic09** need not waste memory space on **shell**.

The **ex** command should always be the last command on a shell input line because any command line following will never be processed.

NOTE: Since this is a built-in **shell** command, it does not appear in the CMDS directory.

For more information see: Section 4.5, Section 4.6, Section 4.9

Examples

```
ex BASIC09
```

```
tsmon /t1&; tsmon /t2&; ex tsmon /term
```

EXBIN

Name

EXBIN — Convert S-Record To Binary File

Synopsis

exbin *path2 path1*

Description

S-Record files are a type of text file that contains records that represent binary data in hexadecimal character form. This Motorola-standard format is often directly accepted by commercial PROM programmers, emulators, logic analyzers and similar devices that are interfaced RS-232 interfaces. It can also be useful for transmitting files over data links that can only handle character-type data; or to convert NitrOS-9 assembler or compiler-generated programs to load on non-NitrOS-9 systems.

"Path1" is assumed to be an S-Record format text file which **exbin** converts to pure binary form on a new file called "path2". The load addresses of each data record must describe continguous data in ascending order.

Exbin does not generate or check for the proper NitrOS-9 module headers or CRC check value required to actually load the binary file. The IDENT or VERIFY commands can be used to check the validity of the modules if they are to be loaded or run. Example:

```
exbin program.S1 cmds/program
```

EXMODE

Name

EXMODE — Examine or Change Device Initialization Mode

Synopsis

exmode *devname* [*arglist*]

Description

exmode is an enhanced verison of the **xmode** utility, and is useful for changing initialization parameters specific to CoCo 3 window descriptors and enhanced ACIA device descriptors.

Exmode is very similar to the **tmode** command. **Tmode** only operates on open paths so its effect is temporary. **Exmode** actually updates the device descriptor so the change persists as long as the computer is running, even if paths to the device are repetitively opened and closed. If **exmode** is used to change parameter(s) and the COBBLER program is used to make a new system disk, the changed parameter will be permanently reflected on the new system disk.

Exmode requires a device name to be given. If no arguments are given, the present values for each parameter are displayed, otherwise, the parameter(s) given in the argument list are processed. Any number of parameters can be given, and are separated by spaces or commas.

Exmode Parameter Names

upc	Upper case only. Lower case characters are automatically converted to upper case.
-upc	Upper case and lower case characters permitted (default).
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).
-bsb	no erase on backspace: echoes single backspace only
bsl	Backspace over line: lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals) (default).
-bsl	No backspace over line: lines are "deleted" by printing a new line sequence (for hard-copy terminals). echo Input characters "echoed" back to terminal (default)
-echo	No echo
lf	Auto line feed on: line feeds automatically echoed to terminal on input and output carriage returns (default).
-lf	Auto line feed off.

pause	Screen pause on: output suspended upon full screen. See "pag" parameter for definition of screen size. Output can be resumed by typing any key.
-pause	Screen pause mode off.
null=n	Set null count: number of null ($00) characters transmitted after carriage returns for return delay. The number is decimal, default = 0.
pag=n	Set video display page length to n (decimal) lines. Used for "pause" mode, see above.
bsp=h	Set input backspace character. Numeric value of character in hexadecimal. Default = 08.
bse=h	Set output backspace character. Numeric value of character in hexadecimal. Default = 08.
del=h	Set input delete line character. Numeric value of character in hexadecimal. Default = 18.
bell=h	Set bell (alert) output character. Numeric value of character in hexadecimal. Default = 07
eor=h	Set end-of-record (carriage return) input character. Numeric value of character in hexadecimal. Default = 0D
eof=h	Set end-of-file input character. Numeric value of character in hexadecimal. Default 1B.
type=h	ACIA initialization value: sets parity, word size, etc. Value in hexadecimal. Default 15
reprint=h	Reprint line character. Numeric value of character in hexadecimal.
dup=h	Duplicate last input line character. Numeric value of character in hexadecimal.
psc=h	Pause character. Numeric value of character in hexadecimal.
abort=h	Abort character (normally **Control**+**C**). Numeric value of character in hexadecimal.
quit=h	Quit character (normally **Control**+**E**). Numeric value of character in hexadecimal.
baud=d	Set baud rate for software-controllable interface. Numeric code for baud rate: 0=110 1=300 2=600 3=1200 4=2400 5=4800 6=9600 7=19200

Examples

```
exmode /TERM -upc lf null=4 bse=1F pause

exmode /T1 pag=24 pause bsl -echo bsp=8 bsl=C

exmode /P baud=3 -if
```

FORMAT

Name

FORMAT — Initialize disk media

Synopsis

format *devname*

Description

This command is used to physically initialize, verify, and establish an initial file structure on a disk. All disks must be formatted before they can be used on an NitrOS-9 system.

NOTE: If the diskette is to be used as a system disk, **OS9gen** or **cobbler** must be run to create the bootstrap after the disk has been formatted.

The formatting process works as follows:

1. The disk surface is physically initialized and sectored.
2. Each sector is read back and verified. If the sector fails to verify after several attempts, the offending sector is excluded from the initial free space on the disk. As the verification is performed, track numbers are displayed on the standard output device.
3. The disk allocation map, root directory, and identification sector are written to the first few sectors of track zero. These sectors *cannot* be defective.

Format will prompt for a disk volume name, which can be up to 32 characters long and may include spaces or punctuation. This name can later be displayed using the FREE command.

For more information see: Section 3.11

FREE

Name

FREE — Display free space remaining on mass-storage device

Synopsis

free *devname*

Description

This command displays the number of unused 256-byte sectors on a device which are available for new files or for expanding existing files. The device name given must be that of a mass-storage multifile device. **Free** also displays the disk's name, creation date, and cluster size.

Data sectors are allocated in groups called "clusters". The number of sectors per cluster depends on the storage capacity and physical characteristics of the specific device. This means that small amounts of free space may not be divisible into as many files. For example, if a given disk system uses 8 sectors per cluster, and a **free** command shows 32 sectors free, a maximum of four new files could be created even if each has only one cluster.

For more information see: Section 3.11

Examples

```
OS9: free
BACKUP DATA DISK created on: 80/06/12
Capacity: 1,232 sectors (1-sector clusters)
1,020 free sectors, largest block 935 sectors

OS9: free /D1
NitrOS-9 Documentation Disk created on: 81/04/13
Capacity: 1,232 sectors (1-sector clusters)
568 Free sectors, largest block 440 sectors
```

HELP

Name

HELP — Displays the usage and syntax of NitrOS-9 commands.

Synopsis

help {*command*}

Description

Provide as argument the command for which you want syntax help. Include as many command names in one **help** line as you wish. The proper form and syntax appears for each valid command you include.

If you do not include a command name, help will show you the list of available topics for you to choose from.

Examples:

```
help ex [ENTER]
Syntax: Ex <modname>
Usage : Chain to the given module
```

```
help me [ENTER]
me: no help available

help [ENTER]
Help available on:
       ASM     ATTR   [...]
```

IDENT

Name

IDENT — Print NitrOS-9 module identification

Synopsis

ident [-opts] *path* [-opts]

Description

This command is used to display header information from NitrOS-9 memory modules. **Ident** displays the module size, CRC bytes (with verification), and for program and device driver modules, the execution offset and the permanent storage requirement bytes. **ident** will print and interpret the type/language and attribute/revision bytes. In addition, **ident** displays the byte immediately following the module name since most Microware-supplied modules set this byte to indicate the module edition.

Ident will display all modules contained in a disk file. If the "-m" option appears, *path* is assumed to be a module in memory.

If the "-v" option is specified, the module CRC is not verified.

The "-x" option implies the pathlist begins in the execution directory.

The "-s" option causes **ident** to display the. following module information on a single line:

Edition byte (first byte after module name)
Type/Language byte
Module CRC
A "." if the CRC verifies correctly, "?" if incorrect. (**Ident** will leave this field blank if the "-v" option appea
Module name

Examples

```
OS9: ident -m  ident
Header for:  Ident              <Module name>
Module size: $06A5    #1701     <Module size>
Module CRC:  $1CE78A (Good)     <Good or Bad>
Hdr parity:  $8B                <Header parity>
Exec. off:   $0222    #546      <Execution offset>
Data size:   $0CA1    #3233     <Permanent storage requirement>
Edition:     $05      #5        <First byte after module name>
Ty/La At/Rv: $11 $81            <Type/Language Attribute/Revision>
```

```
Prog mod, 6809 obj, re-en        <Module type, Language, Attribute>

OS9: ident /d0/os9boot -s
    1 $C0 $A366DC . KernelP2
   83 $C0 $7FC336 . Init
    1 $11 $39BA94 . SysGo
    1 $C1 $402573 . IOMan
    3 $D1 $EE937A . RBF
   82 $F1 $526268 . DD
   82 $F1 $526268 . D0
   82 $F1 $D65245 . D1
   82 $F1 $E32FFE . D2
    1 $D1 $F944D7 . SCF
    2 $E1 $F9FE37 . VDGInt
   83 $F1 $765270 . Term
    2 $D1 $BBC1EE . PipeMan
    2 $E1 $5B2B56 . Piper
   80 $F1 $CC06AF . Pipe
    2 $C1 $248B2C . Clock
    2 $C1 $248B2C . Clock2
    ^  ^     ^    ^ ^
    |  |     |    | |
    |  |     |    | Module name
    |  |     |    CRC check " " if -v, "." if OK, "?" if bad
    |  |     CRC value
    |  Type/Language byte
    Edition byte (first byte after name)
```

INIZ

Name

`INIZ` — Initialize a device.

Synopsis

iniz [`devicename [...]`]

Description

Links the specified device to NitrOS-9, places the device addres in a new device table entry, allocates the memory needed by the device driver, and calls the device driver initialization routine. If the device is already installed, **iniz** does not reinitialize it.

Options:

devicename

is the name of the device drivere you want to initialize. Specify as many device drivers as you wish with one **iniz** command.

Notes:

You can use Iniz in the startup file or at the system startup to initialize devices and allocate their static storage at the top of memory (to reduce memory fragmentation).

Example:

iniz p t2 [ENTER]

initializes the p (printer) and t2 (terminal 2) devices.

IRQS

Name

`IRQS` — Show interrupt polling table

Synopsis

irqs

Description

Irqs displays a list of the system's IRQ polling table. The IRQ polling table contains a list of the service routines for each interrupt handler known by the system.

The irqs display header lists the system name, the NitrOS-9 version number, the maximum number of devices allowed in the device table, and the maximum number of entries in the IRQ table.

KILL

Name

`KILL` — Abort a process

Synopsis

kill *`procID`*

Description

This shell "built in" command sends an "abort" signal to the process having the process ID number specified. The process to be aborted must have the same user ID as the user that executed the command. The **procs** command can be used to obtain the process ID numbers.

NOTE: If a process is waiting for I/O, it may not die until it completes the current I/O operation, therefore, if you **kill** a process and the **procs** command shows it still exists, it is probably waiting for receive a line of data from a terminal before it can die. Since this is a built-in **shell** command, it does not appear in the CMDS directory. For more information see: Section 4.5, Section 5.2, **procs**

Examples

```
kill 5

kill 22

OS9: procs

User # Id pty  state   Mem Primary module
----- --- --- -------- --- --------------
   20  2   0   active   2  Shell <TERM
   20  1   0   waiting  1  Sysgo <TERM
   20  3   0  sleeping 20  Copy <TERM

OS9: kill 3
OS9: procs

User # Id pty  state   Mem Primary module
----- --- --- -------- --- --------------
   20  2   0   active   2  Shell <TERM
   20  1   0   waiting  1  Sysgo <TERM

OS9:
```

LINK

Name

LINK — Link module into memory

Synopsis

link *memory module name*

Description

This command is used to "lock" a previously loaded module into memory. The link count of the module specified is incremented by one each time it is "linked". The **unlink** command is used to "unlock" the module when it is no longer needed.

For more information see: Section 5.4, Section 5.4.1, Section 5.4.2, Section 5.4.3

Examples

```
OS9: LINK edit

OS9: LINK myprogram
```

LIST

Name

`LIST` — List the contents of a text file

Synopsis

list *path* { *path* }

Description

This command copies text lines from the path(s) given as parameters to the standard output path. The program terminates upon reaching the end-of-file of the last input path. If more than one path is specified, the first path will be copied to standard output, the second path will be copied next, etc.

This command is most commonly used to examine or print text files.

For more information see: Section 2.3, Section 3.10.2

Examples

list /d0/startup >/p & (output is redirected to printer)

list /d1/user5/document /d0/myfile /d0/Bob/text

list /term >/p (copy keyboard to printer - use "escape" key to terminate input)

```
OS9: build animals
? cat
? cow
? dog
? elephant
? bird
```

```
? fish
? [RETURN]

OS9: list animals
cat
cow
dog
elephant
bird
fish
```

LOAD

Name

LOAD — Load module(s) from file into memory

Synopsis

load *path*

Description

The path specified is opened and one or more modules is read from it and loaded into memory. The names of the modules are added to the module directory. If a module is loaded that has the same name and type as a module already in memory, the module having the highest revision level is kept.

For more information see: Section 3.10.4, Section 5.4.1, Section 5.4.2

Examples

load new_program

```
OS9:mdir

   Module Directory at 13:36:47
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       REF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         CDS         H1
Sysgo       Clock       Shell       Tsmon       Copy
Mdir

OS9:load edit
OS9:mdir

   Module Directory at 13:37:14
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       REF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         CDS         H1
Sysgo       Clock       Shell       Tsmon       Copy
```

```
Mdir        EDIT
```

LOGIN

Name

LOGIN — Timesharing System Log-In

Synopsis

login

Description

Login is used in timesharing systems to provide log-in security. It is automatically called by the timesharing monitor **tsmon**, or can be used after initial log-in to change a terminal's user.

Login requests a user name and password, which is checked against a validation file. If the information is correct, the user's system priority, user ID, and working directories are set up according to information stored in the file, and the initial program specified in the password file is executed (usually **shell**). If the user cannot supply a correct user name and password after three attempts, the process is aborted. The validation file is called PASSWORD and must be present in the directory /d0/SYS. The file contains one or more variable-length text records, one for each user name. Each record has the following fields, which are delimited by commas:

1. User name (up to 32 characters, may include spaces). If this field is empty, any name will match.

2. Password (up to 32 characters, may include spaces) If this field is omitted, no password is required by the specific use.

3. User index (ID) number (from 0 to 65535, 0 is superuser). This number is used by the file security system and as the system-wide user ID to identify all processes initiated by the user. The system manager should assign a unique ID to each potential user. (See Section 3.9)

4. Initial process (CPU time) priority: 1 - 255 (see Section 5.2)

5. Pathlist of initial execution directory (usually /d0/CMDS)

6. Pathlist of initial data directory (specific user's directory)

7. Name of initial program to execute (usually **shell**). NOTE: This is not a shell command line.

Here's a sample validation file:

```
superuser,secret,0,255,.,.,shell
steve,open sesame,3,128,.,/d1/STEVE,shell
sally,qwerty,10,100,/d0/BUSINESS,/d1/LETTERS,wordprocessor
bob„4,128,.,/d1/BOB,Basic09
```

To use the **login** command, enter:

login

This will cause prompts for the user's name and (optionally) password to be displayed, and if answered correctly, the user is logged into the system. **Login** initializes the user number, working execution directory, working data directory, and executes the initial program specified by the password file. The date, time and process number (which is *not* the same as the user ID, see Section 5.3) are also displayed.

Note: if the shell from which **login** was called will not be needed again, it may be discarded by using the **ex** command to start the **login** command. For example:

ex login

Logging Off the System

To log off the system, the initial program specified in the password file must be terminated. For most programs (including **shell**) this may be done by typing an end of file character (escape) as the first character on a line.

Displaying a "Message-of-the-Day"

If desired, a file named `motd` appearing in the SYS directory will cause **login** to display it's contents on the user's terminal after successful login. This file is not required for **login** to operate.

For more information see: **tsmon**, Section 4.9, Section 3.9, Section 5.3

Examples

```
OS9: login

NitrOS-9/6309 Timesharing System
Level 2 V03.02.01
    2003/12/04 13:02:22

User name?: superuser
Password: secret

Process #07 logged on    2003/12/04 13:03:00
Welcome!
```

MAKDIR

Name

`MAKDIR` — Create directory file

Synopsis

makdir *`path`*

Description

Creates a new directory file acdording to the pathlist given. The pathlist must refer to a parent directory for which the user has write permission.

The new directory is initialized and initially does not contain files except for the . and .. pointers to its parent directory and itself, respectively (see Section 3.8.3). All access permissions are enabled (except sharable).

It is customary (but not mandatory) to capitalize directory names.

For more information see: Section 3.4, Section 3.5, Section 3.6, Section 3.8.3, Section 3.10.5

Examples

makdir /d1/STEVE/PROJECT

makdir DATAFILES

makdir ../SAVEFILES

MDIR

Name

`MDIR` — Display Module Directory

Synopsis

mdir [`-e`]

Description

Displays the present module names in the system module directory, i.e., all modules currently resident in memory. For example:

```
OS9: mdir

 Module Directory at 14:44:35
D0      Pipe    OS9     OS9P2
Init    Boot    DDisk   D1
KBVDIO  TERM    IOMan   RBF
SCF     SysGo   Clock   Shell
PRINTER P       PipeMan Piper
Mdir
```

If the "e" option is given, a full listing of the physical address, size, type, revision level, reentant attribute, user count, and name of each module is displayed. All numbers shown are in hexadecimal.

```
OS9: mdir -e

Module Directory at 10:55:04
```

```
ADDR SIZE TY RV AT UC   NAME
---- ---- -- -- -- -- --------
C305   2F F1  1 R     D0
F059  7EB C1  1 R     OS9
F852  4F4 C1  1 R     OS9P2
FD46   2E CO  1 R     INIT
C363  798 E1  1 R   2 KBVDIO
CAFB   38 F1  1 R   2 TERM
```

Caution

Many of the modules listed by **mdir** are NitrOS-9 system modules and *not* executable as programs: always check the module type code before running a module if you are not familiar with it!

For more information see: Section 5.4.1

MERGE

Name

MERGE — Copy and Combine Files to Standard Output

Synopsis

merge *path* {*path*}

Description

This command copies multiple input files specified by the pathlists given as parameters to the standard output path. it is commonly used to combine several files into a single output file. Data is copied in the order the pathlists are given. **Merge** does no output line editing (such as automatic line feed). The standard output is generally redirected to a file or device.

Examples

```
OS9: merge file1 file2 file3 file4 >combined.file

OS9: merge compile.list asm.list >/printer
```

MFREE

Name

MFREE — Display Free System RAM

Synopsis

mfree

Description

Displays a list of which areas of memory are not presently in use and available for assignment. The address and size of each free memory block are displayed. The size is given as the number of 256-byte pages. This information is useful to detect and correct memory fragmentation (see Section 5.4.3).

For more information see: Section 5.4, Section 5.4.3

Examples

```
OS9: mfree

 Address  pages
--------- -----
 700- 7FF    1
 B00-AEFF  164
B100-B1FF    1

Total pages free = 166
```

OS9GEN

Name

OS9GEN — Build and Link a Bootstrap File

Synopsis

os9gen *device name*

Description

OS9Gen is used to create and link the OS9Boot file required on any disk from which OS-9 is to be bootstrapped. **OS9Gen** is used to add modules to an existing boot or to create an entirely new boot file. If an exact copy of the existing OS9Boot file is desired, the **cobbler** command should be used instead.

The name of the device on which the OS9Boot file is to be installed is passed to **OS9Gen** as a command line parameter. **OS9Gen** then creates a working file called TempBoot on the device specified. Next it reads file names (pathlists) from its standard input, one pathlist per line. Every file named is opened and copied to TempBoot. This is repeated until end-of-file or a blank line is reached on **OS9Gen**'s standard input. All boot files must contain the OS-9 component modules listed in section Section 6.1.

After all input files have been copied to TempBoot, the old OS9Boot file, if present, is deleted. TempBoot is then renamed to OS9Boot, and its starting address and size is linked in the disk's Identification Sector (LSN 0) for use by the OS-9 bootstrap firmware.

WARNING: Any OS9Boot file must be stored in physically contiguous sectors. Therefore, **OS9Gen** is normally used on a freshly formatted disk. If the OS9Boot file is fragmented, **OS9Gen** will print a warning message indicated the disk cannot be used to bootstrap OS-9.

The list of file names given to **OS9Gen** can be entered from a keyboard, or **OS9Gen**'s standard input may be redirected to a text file containing a list of file names (pathlists) . If names are entered manually, no prompts are given, and the end-of-file key (usually ESCAPE) or a blank line is entered after the line containing the last pathlist.

For more information see: Chapter 6, Section 6.1, Section 6.6

Examples

To manually install a boot file on device "d1" which is an exact copy of the OS9Boot file on device "d0":

```
OS9: os9gen /d1          (run OS9Gen)
/d0/os9boot              (enter file to be installed)
[ESCAPE]                 (enter end-of-file)
```

To manually install a boot file on device "d1" which is a copy of the OS9Boot file on device "d0" with the addition of modules stored in the files /d0/tape.driver and /d2/video.driver:

```
OS9: os9gen /d1          (run OS9Gen)
/d0/os9boot              (enter main boot file name)
/d0/tape.driver          (enter name of first file to be added)
/d2/video.driver         (enter name of second file to be added)
[ESCAPE]                 (enter end-of-file)
```

As above, but automatically by redirecting **OS9Gen** standard input:

```
OS9: build /d0/bootlist  (use build to create file bootlist)
? /d0/os9boot            (enter first file name)
? /d0/tape.driver        (enter second file name)
? /d2/video.driver       (enter third file name)
? [RETURN]               (terminate build)
OS9: os9gen /d1 </d0/bootlist  (run OS9gen with redirected input)
```

PRINTERR

Name

PRINTERR — Print Full Text Error Messages

Synopsis

printerr

Description

This command replaces the basic OS-9 error printing routine (F$Perr service request) which only prints error code numbers, with a routine the reads and displays textual error messages from the file `/d0/SYS/errmsg`. Printerr's effect is system-wide.

A standard error message file is supplied with OS-9. This file can be edited or replaced by the system manager. The file is a normal text file with variable length line. Each error message line begins with the error number code (in ASCII characters), a delimiter, and the error message text. The error messages need not be in any particular order. Delimiters are spaces or any character numerically lower then $20. Any line having a delimiter as its first character is considered a continuation of the previous line(s) which permits multi-line error messages.

Warning

Once the **printerr** command has been used, it can not be undone. Once installed, the **printerr** module should not be unlinked. **Printerr** uses the current user's stack for an I/O buffer, so users are encouraged to reserve reasonably large stacks.

For more information see: Section 4.7, Section 6.2.

Examples

```
OS9: printerr
```

PROCS

Name

`PROCS` — Display Processes

Synopsis

procs [`-e`]

Description

Displays a list of processes running on the system. Normally only processes having the user's ID are listed, but if the "-e" option is given, processes of all users are listed. The display is a "snapshot" taken at the instant the command is executed: processes can switch states rapidly, usually many times per second.

PROCS shows the user and process ID numbers, priority, state (process status), memory size (in 256 byte pages), primary program module, and standard input path.

For more information see: Section 5.1, Section 5.2, Section 5.3

Examples

Level One Example:

```
User# Id pty  state   Mem Primary module
----- --- --- -------- --- --------------
   0   2   0  active    2 Shell
   0   1   0  waiting   1 SysGo
   1   3   1  waiting   2 Tsmon
   1   4   1  waiting   4 Shell
   1   5   1  active   64 Basic09
```

PWD/PXD

Name

`PWD/PXD` — Print Working Directory / Print Execution Directory

Synopsis

pwd

pxd

Description

Pwd displays a pathlist that shows the path from the root directory to the user's current data directory. It can be used by programs to discover the actual physical location of files, or by humans who get lost in the file system. **Pxd** is identical except that is shows the pathlist of the user's current execution directory.

Examples

```
OS9: chd /D1/STEVE/TEXTFILES/MANUALS
OS9: pwd
/D1/STEVE/TEXTFILES/MANUALS
OS9: chd ..
OS9: pwd
/D1/STEVE/TEXTFILES
OS9: chd ..
OS9: pwd
/D1/STEVE

OS9: pxd
/D0/CMDS
```

RENAME

Name

RENAME — Change file name

Synopsis

rename *path new name*

Description

Gives the mass storage file specified in the pathlist a new name. The user must have write permission for the file to change its name. It is not possible to change the names of devices, ., or ..

Examples

rename blue purple

rename /D3/user9/test temp

```
OS9: dir

   Directory of .  16:22:53
myfile          animals

OS9:rename animals cars
OS9:dir

   Directory of .  16:23:22
myfile          cars
```

RUNB

Name

RUNB — BASIC09 run time package

Synopsis

runb *i-code module*

Description

BASIC09 run time package

Once one or more BASIC09 procedures are debugged to the programmer's satisfaction, they can be "packed" or converted permanently to the bytecode form.

Packed BASIC09 procedures are in fact OS-9 modules, and the OS-9 shell recognizes them as I-code and passes them off to the virtual machine emulator RunB for execution. RunB avoids a great deal of the overhead of the typical interpreted BASICs of the day -- not to mention that one can do integer calculations where appropriate rather than doing everything in floating point -- so that BASIC09 programs run very quickly in comparison with interpreted BASICs.

SAVE

Name

SAVE — Save memory module(s) on a file

Synopsis

save *path modname* {*modname*}

Description

Creates a new file and writes a copy of the memory module(s) specified on to the file. The module name(s) must exist in the module directory when saved. The new file is given access permissions for all modes except public write.

Note: **save**'s default directory is the current data directory. Executable modules should generally be saved in the default execution directory.

Examples

save wordcount wcount

save /d1/mathpack add sub mul div

SETIME

Name

SETIME — Activate and set system clock

Synopsis

setime [y,m,d,h,m,s]

Description

This command sets the system date and time, then activates the real time clock. The date and time can be entered as parameters, or if no parameters are given, **setime** will issue a prompt. Numbers are one or two decimal digits using space, colon, semicolon or slash delimiters. OS-9 system time uses the 24 hour clock, i.e., 1520 is 3:20 PM.

> **Important:** This command must be executed before OS-9 can perform multitasking operations. If the system does not have a real time clock this command should still be used to set the date for the file system.

> **Systems With Battery Backed up Clocks: Setime** should still be run to start time-slicing, but only the *year* need be given, the date and time will be read from the clock.

Examples

```
OS9: setime 82,12,22,1545 (Set to: Dec. 12, 1981, 3:45 PM)

OS9: setime 821222 154500 (Same as above)

OS9: setime 82            (For system with battery-backup clock)
```

SETPR

Name

SETPR — Set Process Priority

Synopsis

setpr *procID number*

Description

This command changes the CPU priority of a process. It may only be used with a process having the user's ID. The process number is a decimal number in the range of 1 (lowest) to 255. The **procs** command can be used to obtain process ID numbers and present priority.

NOTE: This command does not appear in the CMDS directory as it is built-in to the **shell**.

For more information see: Section 5.1, **procs**

Examples

setpr 8 250 (change process #8 priority to 250)

```
OS9: procs

User # Id pty  state   Mem Primary module
----- --- --- -------- --- --------------
    0   3   0 waiting    2 Shell <TERM
    0   2   0 waiting    2 Shell <TERM
    0   1   0 waiting    1 Sysgo <TERM


OS9: setpr 3 128
OS9: procs

User # Id pty  state   Mem Primary module
----- --- --- -------- --- --------------
    0   3 128 active     2 Shell <TERM
    0   2   0 waiting    2 Shell <TERM
    0   1   0 waiting    1 Sysgo <TERM
```

SHELL

Name

SHELL — OS-9 Command Interpreter

Synopsis

shell *arglist*

Description

The **shell** is OS-9's command interpreter program. It reads data from its standard input path (the keyboard or a file), and interprets the data as a sequence of commands. - The basic function of the shell is to initiate and control execution of other OS-9 programs.

The shell reads and interprets one text line at a time from the standard input path. After interpretation of each line it reads another until an end-of-file condition occurs, at which time it terminates itself. A special case is when the shell is called from another program, in which case it will take the parameter area (rest of the command line) as its first line of input. If this command line consists of "built in" commands only, more lines will be read and processed; otherwise control will return to the calling program after the single command line is processed.

The rest of this description is a technical specification of the shell syntax. Use of the **shell** is described fully in Chapters 2 and 4 of this manual.

Shell Input Line Formal Syntax

```
pgm line := pgm {pgm}
pgm := [params] [ name [modif] [pgm params] [modif] ] [sep]

Program Specifications

name := module name
            := pathlist
            := ( pgm list )

Parameters

params:= param { delim param }
delim := space or comma characters
param := ex name [modif] chain to program specified
          := chd pathlist    change working directory
          := kill procID     send abort signal to process
          := setprprocID pty change process priority
          := chx pathlist    change execution directory
          := w                                    wait for any process to die
          := p                                    turn "OS9:" prompting on
          := -p                                   turn prompting off
          := t                                    echo input lines to std output
          := -t                                   don't echo input lines
          := -x                                   dont abort on error
          := x                                    abort on error
          := * text          comment line: not processed
sep   := ;     sequential execution separator
          := &     concurrent execution separator
          := !     pipeline separator
          := cr end-of-line (sequential execution separator)


Modifiers

modif := mod { delim mod }
mod   := < pathlist redirect standard input
         := > pathlist redirect standard output
         := >> pathlist redirect standard error output
         := # integer set process memory size in pages
         := # integer K   set program memory size in 1K increments
```

SLEEP

Name

SLEEP — Suspend process for period of time

Synopsis

sleep *tickcount*

Description

This command puts the user's process to "sleep" for a number of clock ticks. It is generally used to generate time delays or to "break up" CPU-intensive jobs. The duration of a tick is 16.66 milliseconds.

A tick count of 1 causes the process to "give up" its current time slide. A tick count of zero causes the process to sleep indefinitely (usually awakened by a signal)

Examples

```
OS9: sleep 25
```

TEE

Name

`TEE` — Copy standard input to multiple output paths

Synopsis

tee {*path*}

Description

This command is a filter (see Section 4.3.3) that copies all text lines from its standard input path to the standard output path *and* any number of additional output paths whose pathlists are given as parameters.

The example below uses a pipeline and **tee** to simultaneously send the output listing of the **dir** command to the terminal, printer, and a disk file:

```
dir e ! tee /printer /d0/dir.listing
```

The following example sends the output of an assembler listing to a disk file and the printer:

```
asm pgm.src l ! tee pgm.list >/printer
```

The example below "broadcasts" a message to four terminals:

```
echo WARNING System down in 10 minutes ! tee /t1 /t2 /t3 /t4
```

TMODE

Name

`TMODE` — Change terminal operating mode

Synopsis

tmode [*.pathnum*] [*arglist*]

Description

This command is used to display or change the operating parameters of the user's terminal.

If no arguments are given, the present values for each parameter are displayed, otherwise, the parameter(s) given in the argument list are processed. Any number of parameters can be. given, and are separated by spaces or commas. A period and a number can be used to optionally specify the path number to be affected. If none is given, the standard input path is affected.

NOTE: If this command is used in a shell procedure file, the option "*.path num*" must be used to specify one of the standard output paths (0, 1 or 2) to change the terminal's operating characteristics. The change will remain in effect until the path is closed. To effect a permanent change to a device characteristic, the device descriptor must be changed.

This command can work only if a path to the file/device has already been opened. You may alter the device descriptor to set a device's initial operating parameter (see the System Programmer's Manual).

upc	Upper case only. Lower case characters are automatically converted to upper case.
-upc	Upper case and lower case characters permitted (default).
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).
-bsb	no erase on backspace: echoes single backspace only
bsl	Backspace over line: lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals) (default).
-bsl	No backspace over line: lines are "deleted" by printing a new line sequence (for hard-copy terminals). echo Input characters "echoed" back to terminal (default)
-echo	No echo
lf	Auto line feed on: line feeds automatically echoed to terminal on input and output carriage returns (default).
-lf	Auto line feed off.
pause	Screen pause on: output suspended upon full screen. See "pag" parameter for definition of screen size. Output can be resumed by typing any key.
-pause	Screen pause mode off.
null=n	Set null count: number of null ($00) characters transmitted after carriage returns for return delay. The number is decimal, default = 0.
pag=n	Set video display page length to n (decimal) lines. Used for "pause" mode, see above.
bsp=h	Set input backspace character. Numeric value of character in hexadecimal. Default = 08.

bse=h	Set output backspace character. Numeric value of character in hexadecimal. Default = 08.
del=h	Set input delete line character. Numeric value of character in hexadecimal. Default = 18.
bell=h	Set bell (alert) output character. Numeric value of character in hexadecimal. Default = 07
eor=h	Set end-of-record (carriage return) input character. Numeric value of character in hexadecimal. Default = 0D
eof=h	Set end-of-file input character. Numeric value of character in hexadecimal. Default 1B.
type=h	ACIA initialization value: sets parity, word size, etc. Value in hexadecimal. Default 15
reprint=h	Reprint line character. Numeric value of character in hexadecimal.
dup=h	Duplicate last input line character. Numeric value of character in hexadecimal.
psc=h	Pause character. Numeric value of character in hexadecimal.
abort=h	Abort character (normally **Control**+**C**). Numeric value of character in hexadecimal.
quit=h	Quit character (normally **Control**+**E**). Numeric value of character in hexadecimal.
baud=d	Set baud rate for software-controllable interface. Numeric code for baud rate: 0=110 1=300 2=600 3=1200 4=2400 5=4800 6=9600 7=19200

Examples

```
tmode -upc lf null=4 bselF pause

tmode pag=24 pause bsl -echo bsp=8 bsl=C
```

NOTE: If you use **tmode** in a procedure file, it will be necessary to specify one of the standard output paths (.1 or .2) since the shell's standard input path will have been redirected to the disk file (**Tmode** can be used on an SCFMAN-type devices only). Example:

```
tmode .1 pag=24          (set lines/page on standard output)
```

TSMON

Name

`TSMON` — Timesharing monitor

Synopsis

tsmon [*pathlist*]

Description

This command is used to supervise idle terminals and initiate the login sequence in timesharing applications. If a pathlist is given, standard I/O paths are opened for the device. When a carriage return is typed, **tsmon** will automatically call the **login** command. If the login fails because the user could not supply a valid user name or password, it will return to **tsmon**.

Note: The **login** command and its password file must be present for **tsmon** to work correctly (see the **login** command description).

Logging Off the System

Most programs will terminate when an end of file character (escape) is entered as the first character on a command line. This will log you off of the system and return control to **tsmon**.

For more information see: Section 4.9, **login**

Examples

```
OS9:tsmon /t1&
&005
```

TUNEPORT

Name

TUNEPORT — Tune the printer port on the Color Computer

Synopsis

tuneport [-s=value]

Description

This command lets you test and set delay loop values for the current baud rate and select the best value for your printer (/p) or terminal (/t1).

Examples

```
TUNEPORT /P [ENTER]
```

Provides a text operation for your printer. After a short delay, TUNEPORT displays the current baud rate and sends data to the printer to test if it is working properly. The program then displays the current delay value and asks for a new value. Enter a decimal delay value and press [ENTER]. Again, test data is sent to the printer as a test. Continue this process until you find the best value. When you are satisfied, press [ENTER] instead of entering a value at the prompt. A closing message displays your new value.

Use the same process to set a new delay loop value for /t1 terminal

```
tuneport /p -s=225 [ENTER]
```

Sets the delay loop value for your printer at 225. Use such a command on future system boots to set the optimum delay value determined with the TUNEPORT test function. Then, using OS9GEN or COBBLER, generate a new boot file for your system diskette. You can also use TUNEPORT in your system startup file to set the value using the -s option.

UNLINK

Name

UNLINK — Unlink memory module

Synopsis

unlink *modname* {*modname*}

Description

Tells OS-9 that the memory module(s) named are no longer needed by the user. The module(s) may or may not be destroyed and their memory reassigned, depending on if in use by other processes or user, whether resident in ROM or RAM, etc.

It is good practice to unload modules whenever possible to make most efficient use of available memory resources.

Warning: never unlink a module you did not load or link to.

For more information see: Section 5.4, Section 5.4.1, Section 5.4.2

Examples

```
unlink pgml pgm5 pgm99


OS9: mdir

   Module Directory at 11:26:22
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       RBF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         Sysgo       Clock
Shell       Tsmon       Edit
```

```
OS9: unlink edit
OS9: mdir

   Module Directory at 11:26:22
DCB4        D0          D1          D2          D3
OS9P2       INIT        OS9         IOMAN       RBF
SCF         ACIA        TERM        T1          T2
T3          P           PIA         Sysgo       Clock
Shell       Tsmon
```

VERIFY

Name

VERIFY — Verify or update module header and CRC

Synopsis

verify [-u]

Description

This command is used to verify that module header parity and CRC value of one or more modules on a file (standard input) are correct. Module(s) are read from standard input, and messages will be sent to the standard error path.

If the -u (update) option is specified, the module(s) will be copied to the standard output path with the module's header parity and CRC values replaced with the computed values. A message will be displayed to indicate whether or not the module's values matched those computed by **verify**.

If the option is NOT specified, the module will not be copied to standard output. **Verify** will only display a message to indicate whether or not the module's header parity and CRC matched those which were computed.

Examples

```
OS9: verify <EDIT >NEWEDIT

Module's header parity is correct.
Calculated CRC matches module's.

OS9: verify <myprogram1 >myprogram2

Module's header parity is correct.
CRC does not match.

OS9: verify <myprogram2

Module's header parity is correct.
Calculated CRC matches module's.

OS9: verify -u <module >temp
```

XMODE

Name

XMODE — Examine or Change Device Initialization Mode

Synopsis

xmode *devname* [*arglist*]

Description

This command is used to display or change the initialization parameters of any SCF-type device such as the video display, printer, RS232 port, etc. A common use is to change baud rates, control key definitions, etc.

Xmode is very similar to the **tmode** command. **Tmode** only operates on open paths so its effect is temporary. **Xmode** actually updates the device descriptor so the change persists as long as the computer is running, even if paths to the device are repetitively opened and closed. If **xmode** is used to change parameter(s) and the **cobbler** program is used to make a new system disk, the changed parameter will be permanently reflected on the new system disk.

Xmode requires a device name to be given. If no arguments are given, the present values for each parameter are displayed, otherwise, the parameter(s) given in the argument list are processed. Any number of parameters can be given, and are separated by spaces or commas.

XMODE Parameter Names

upc	Upper case only. Lower case characters are automatically converted to upper case.
-upc	Upper case and lower case characters permitted (default).
bsb	Erase on backspace: backspace characters echoed as a backspace-space-backspace sequence (default).
-bsb	no erase on backspace: echoes single backspace only
bsl	Backspace over line: lines are "deleted" by sending backspace-space-backspace sequences to erase the same line (for video terminals) (default).
-bsl	No backspace over line: lines are "deleted" by printing a new line sequence (for hard-copy terminals). echo Input characters "echoed" back to terminal (default)
-echo	No echo
lf	Auto line feed on: line feeds automatically echoed to terminal on input and output carriage returns (default).
-lf	Auto line feed off.

pause	Screen pause on: output suspended upon full screen. See "pag" parameter for definition of screen size. Output can be resumed by typing any key.
-pause	Screen pause mode off.
null=n	Set null count: number of null ($00) characters transmitted after carriage returns for return delay. The number is decimal, default = 0.
pag=n	Set video display page length to n (decimal) lines. Used for "pause" mode, see above.
bsp=h	Set input backspace character. Numeric value of character in hexadecimal. Default = 08.
bse=h	Set output backspace character. Numeric value of character in hexadecimal. Default = 08.
del=h	Set input delete line character. Numeric value of character in hexadecimal. Default = 18.
bell=h	Set bell (alert) output character. Numeric value of character in hexadecimal. Default = 07
eor=h	Set end-of-record (carriage return) input character. Numeric value of character in hexadecimal. Default = 0D
eof=h	Set end-of-file input character. Numeric value of character in hexadecimal. Default 1B.
type=h	ACIA initialization value: sets parity, word size, etc. Value in hexadecimal. Default 15
reprint=h	Reprint line character. Numeric value of character in hexadecimal.
dup=h	Duplicate last input line character. Numeric value of character in hexadecimal.
psc=h	Pause character. Numeric value of character in hexadecimal.
abort=h	Abort character (normally **Control**+**C**). Numeric value of character in hexadecimal.
quit=h	Quit character (normally **Control**+**E**). Numeric value of character in hexadecimal.
baud=d	Set baud rate for software-controllable interface. Numeric code for baud rate: 0=110 1=300 2=600 3=1200 4=2400 5=4800 6=9600 7=19200

Examples

```
xmode /TERM -upc lf null=4 bse=1F pause

xmode /T1 pag=24 pause bsl -echo bsp=8 bsl=C

xmode /P baud=3 -if
```

Appendix A. OS-9 Error Codes

The error codes are shown in both hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

HEX	DEC	
$C8	200	PATH TABLE FULL - The file cannot be opened because the system path table is currently full.
$C9	201	ILLEGAL PATH NUMBER - Number too large or for non-existant path.
$CA	202	INTERRUPT POLLING TABLE FULL
$CB	203	ILLEGAL MODE - attempt to perform I/O function of which the device or file is incapable.
$CC	204	DEVICE TABLE FULL - Can't add another device
$CD	205	ILLEGAL MODULE HEADER - module not loaded because its sync code, header parity, or CRC is incorrect.
$CE	206	MODULE DIRECTORY FULL - Can't add another module
$CF	207	MEMORY FULL - Level One: not enough contiquous RAM free. Level Two: process address space full
$D0	208	ILLEGAL SERVICE REQUEST - System call had an illegal code number.
$D1	209	MODULE BUSY - non-sharable module is in use by another process.
$D2	210	BOUNDARY ERROR - Memory allocation or deallocation request not on a page boundary.
$D3	211	END OF FILE - End of file encountered on read.
$D4	212	RETURNING NON-ALLOCATED MEMORY - attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT - device has damaged file structure.
$D6	214	NO PERMISSION - file attributes do not permit access requested.
$D7	215	BAD PATH NAME - syntax error in pathlist (illegal character, etc.).
$D8	216	PATH NAME NOT FOUND - can't find pathlist specified.
$D9	217	SEGMENT LIST FULL - file is too fragmented to be expanded further.
$DA	218	FILE ALREADY EXISTS - file name already appears in current directory.
$DB	219	ILLEGAL BLOCK ADDRESS - device's file structure has been damaged.
$DC	220	ILLEGAL BLOCK SIZE - device's file structure has been damaged.
$DD	221	MODULE NOT FOUND - request for link to module not found in directory.
$DE	222	SECTOR OUT OF RANGE - device file structure damaged or incorrectly formatted.

HEX	DEC	
$DF	223	SUICIDE ATTEMPT - request to return memory where your stack is located.
$E0	224	ILLEGAL PROCESS NUMBER - no such process exists.
$E2	226	NO CHILDREN - can't wait because process has no children.
$E3	227	ILLEGAL SWI CODE - must be 1 to 3.
$E4	228	PROCESS ABORTED - process aborted by signal code 2.
$E5	229	PROCESS TABLE FULL - can't fork now.
$E6	230	ILLEGAL PARAMETER AREA - high and low bounds passed in fork call are incorrect.
$E7	231	KNOWN MODULE - for internal use only.
$E8	232	INCORRECT MODULE CRC - module has bad CRC value.
$E9	233	SIGNAL ERROR - receiving process has previous unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE - unable to locate module.
$EB	235	BAD NAME - illegal name syntax
$EC	236	BAD HEADER - module header parity incorrect
$ED	237	RAM FULL - no free system RAM available at this time
$EE	238	UNKNOWN PROCESS ID - incorrect process ID number
$EF	239	NO TASK NUMBER AVAILABLE - all task numbers in use

A.1. Device Driver Errors

The following error codes are generated by I/O device drivers, and are somewhat hardware dependent. Consult manufacturer's hardware manual for more details.

$F0	240	UNIT ERROR - device unit does not exist.
$F1	241	SECTOR ERROR - sector number is out of range.
$F2	242	WRITE PROTECT - device is write protected.
$F3	243	CRC ERROR - CRC error on read or write verify.
$F4	244	READ ERROR - Data transfer error during disk read operation, or SCF (terminal) input buffer overrun.
$F5	245	WRITE ERROR - hardware error during disk write operation.
$F6	246	NOT READY - device has "not ready" status.
$F7	247	SEEK ERROR - physical seek to non-existant sector.
$F8	248	MEDIA FULL - insufficient free space on media.
$F9	249	WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive)
$FA	250	DEVICE BUSY - non-sharable device is in use
$FB	251	DISK ID CHANGE - Media was changed with files open
$FC	252	RECORD IS LOCKED-OUT - Another process is accessing the requested record.
$FD	253	NON-SHARABLE FILE BUSY - Another process is accessing the requested file.

Appendix B. VDG Display System Functions

B.1. The Video Display Generator

NitrOS-9 allows the VDG display to be used in alphanumeric, semigraphic, and graphics modes. There are many built-in functions to control the display, which are activated by used of various ASCII control character. Thus, these functions are available for use by software written in any language using standard output statements (such as "PRINT" in BASIC). The Color Computer's Basic09 language has a Graphics Interface Module that can automatically generate these codes using Basic09 RUN statements.

The display system has two display modes: Alphanumeric ("Alpha") mode and Graphics mode. The Alphanumeric mode also includes "semigraphic" box-graphics. The Color Computer's display system uses a separate - memory area for each display mode so operations on the Alpha display do not affect the Graphics display, and visa-versa. Either display can be selected under software control.

8-bit characters sent to the display system are interpreted according to their numerical value, as shown in the chart below.

Character Range (Hex)	Mode/Used For
00 - 0E	Alpha Mode - cursor and screen control
0F - 1B	Graphics Mode - drawing and screen control
1C - 20	Not used
20 - SF	Alpha Mode - upper case characters
60 - 7F	Alpha Mode - lower case characters
80 - FF	Alpha Mode - Semigraphic patterns

The graphics and alphanumeric functions are handled by the OS-9 device driver module called "CCIO".

B.2. Alpha Mode Display

This is the "standard" operational mode. It is used to display alphanumeric characters and semigraphic box graphics, and simulates the operation of a typical computer terminal with functions for scrolling, cursor positioning, clear screen, line delete, etc.

Each 8-bit character is assumed to be an ASCII character and is displayed if its high order bit (sign bit) is cleared. Lower case letters are displayed in reverse video. If the high order bit of the character is set it is assumed to be a "Semigraphic 6" graphics box. See the Color Computer manual for an explanation of semigraphics functions.

Table B-1. Alpha Mode Command Codes

Control Code	Name/Function
01	HOME - return cursor to upper left hand corner of screen
02	CURSOR XY - move cursor to character X of line Y. The binary value minus 32 of the two characters following the control character are used as the X and Y coordinates. For example, to position the cursor at character 5 of line 10, you must give X=37 and Y42
03	ERASE LINE - erases all characters on the cursor's line.

Control Code	Name/Function
06	CURSOR RIGHT - move cursor right one character position
08	CURSOR LEFT - move cursor left one character position
09	CURSOR UP - move cursor up one line
10	CURSOR DOWN (linefeed) move cursor down one line
12	CLEAR SCREEN - erase entire screen and home cursor
13	RETURN - return cursor to leftmost character of line
14	DISPLAY ALPHA - switch screen from graphic mode to alpha numeric mode

B.3. Graphics Mode Display

This mode is used to display high-resolution 2- or 4-color graphics, and it includes commands to: set color; plot and erase individual points; draw and erase lines; position the graphics cursor; and draw circles.

The DISPLAY GRAPHICS command must be executed before any other graphics mode command is used. It causes the graphics screen to be displayed and sets a current display format and color. The Li.u.t time the DISPLAY GRAPHICS command is given, a 6144 byte display memory is allocated by OS-9, so there must be at least this much continuous free memory available (the OS-9 "MFREE" command can be used to check free memory). This memory is retained until the END GRAPHICS command is given, even if the program that initiated Graphics mode finishes, so it important that the END GRAPHICS command be used to give up the display memory when Graphics mode is no longer needed.

Graphics mode supports two basic formats: Two-Color which has 256 horizontal by 192 vertical points (G6R mode); and Four Color which has 128 horizontal by 192 vertical points (G6C mode). Two color sets are available in either mode. Regardless of the resolution of the format selected, all Graphics mode commands use a 256 by 192 point coordinate system. The X and Y coordinates are always positive numbers which assume that point 0,0 is the lower lefthand corner of the screen.

An invisible Graphics Cursor is used by many command to reduce the amount of output required to generate graphics. This cursor can be explicitly set to any point using the SET GRAPHICS CURSOR command. Also, all other commands that include X,Y coordinates (such as SET POINT) move the graphics cursor to the specified position.

Table B-2. Graphics Mode Selection Codes

Code	Format
00	256 x 192 two-color graphics
01	128 x 192 four-color graphics

Table B-3. Color Set and Current Foreground Color Selection Codes

		Two Color Format		Four Color Format	
	Char	Background	Foreground	Background	Foreground
Color Set 1	00	Black	Black	Green	Green
	01	Black	Green	Green	Yellow

		Two Color Format		Four Color Format	
	Char	Background	Foreground	Background	Foreground
	02			Green	Blue
	03			Green	Red
Color Set 2	04	Black	Black	Buff	Buff
	05	Black	Buff	Buff	Cyan
	06			Buff	Magenta
	07			Buff	Orange
Color Set 3*	08			Black	Black
	09			Black	Dark Green
	10			Black	Med. Green
	11			Black	Light Green
Color Set 4*	12			Black	Black
	13			Black	Green
	14			Black	Red
	15			Black	Buff

* Color sets 3 and 4 not available on PAL video system (European) models. These color sets work only with NTSC (U.S., Canada, Japan) models.

Table B-4. Graphics Mode Control Commands

Control Code	Name/Function
15	DISPLAY GRAPHICS - switches screen to graphics mode. This command must be given before any other graphics commands are used. The first time this command is given, a 6K byte display buffer is assigned. If 6K of contiguous memory is not available an error is returned. This command is followed by two characters which specify the graphics mode and current color/color set, respectively.
16	PRESET SCREEN - presets entire screen to color code passed in next character.
17	SET COLOR - selects foreground color (and color set) passed in next character, but does not change graphics mode.
18	QUIT GRAPHICS - disables graphics mode and returns the 6K byte graphics memory area to OS-9 for other use, and switches to alpha mode.
19	ERASE GRAPHICS - erases all points to background color and homes graphics cursor to the desired position.
20	HOME GRAPHICS CURSOR - moves graphics cursor to coordinates 0,0 (lower left hand corner).
21	SET GRAPHICS CURSOR - moves graphics cursor to given coordinates X,Y. The binary value of the two characters that immediately follow are used as the X and Y values, respectively.

Control Code	Name/Function
22	DRAW LINE - draws a line of the current foreground color from the current graphics cursor position to the given X,Y coordinates. The binary value of the two characters that immediately follow are used as the X and Y values, respectively. The graphics cursor is moved to the end point of the line.
23	ERASE LINE - same as DRAW LINE except the line is "drawn" in the current background color, thus erasing the line.
24	SET POINT - sets the pixel-at point X,Y to the current foreground color. The binary value of the two characters that immediately follow are used as the x and Y values, respectively. The graphics cursor is moved to the point Set.
25	ERASE POINT - same as DRAW POINT except the point is "drawn" in the current background color, thus erasing the point.
26	DRAW CIRCLE - draws a circle of the current foreground color with its center at the current graphics cursor position using a radius R which is obtained using the binary value of the next character. The graphics cursor position is not affected by this command.

B.4. Get Status Commands

The Color Computer I/O driver includes OS-9 Get Status commands that return the display status and joystick values, respectively. These are accessable via the Basic09 Graphics Interface Module, or by the assembly language system calls listed below:

GET DISPLAY STATUS:

Calling Format	`lda #1 (path number)`
	`ldb #SS.DStat (Getstat code $12)`
	`os9 I$GSTT call OS-9`

Passed	nothing
Returns	X = address of graphics display memory Y = graphics cursor address x=MSB y =LSB A = color code of pixel at cursor address

GET JOYSTICK VALUES:

Calling Format	`lda #1 (path number)`
	`ldb #SS.Joy (Getstat code $13)`
	`os9 I$GSTT call OS-9`

Passed	X = 0 for right joystick; 1 for left joystick
Returns	X = selected joystick x value (0-63) Y = selected joystick y value (0-63) A = $FF if fire button on; $00 if off

Table B-5. Display Control Codes Condensed Summary

1st Byte	2nd Byte	3rd Byte	Function
00			Null

1st Byte	2nd Byte	3rd Byte	Function
01			Home Alpha Cursor
02	Column+32	Row+32	Position Alpha Cursor
03			Erase Line
06			Cursor Right
08			Cursor Left
09			Cursor Up
10			Cursor Down
12			Clear Screen
13			Carriage Return
14			Select Alpha Mode
15	Mode	Color Code	Select Graphics Mode
16	Color Code		Preset Screen
17	Color Code		Select Color
18		Quit Graphics Mode	
19		Erase Screen	
20		Home Graphics Cursor	
21	X Coord	Y Coord	Move Graphics Cursor
22	X Coord	Y Coord	Draw Line to X/Y
23	X Coord	Y Coord	Erase Line to X/Y
24	X Coord	Y Coord	Set Point at X/Y
25	X Coord	Y Coord	Clear Point at X/Y
26	Radius		Draw Circle

Appendix C. Key Definitions With Hexadecimal Values

NORM	SHFT	CTRL	NORM	SHFT	CTRL	NORM	SHFT	CTRL
0 30	0 30	--	@ 40	’ 60	NUL 00	P 50	p 70	DLE 10
1 31	1 21	| 7C	A 41	a 61	SOH 01	Q 51	q 71	DC1 11
2 32	" 22	00	B 42	b 62	STX 02	R 52	r 72	DC2 12
3 33	# 23	- 7E	C 43	c 63	ETX O3	S 53	s 73	DC3 13
4 34	$ 24	00	0 44	d 64	EOT 04	T 54	t 74	DC4 14
5 35	% 25	00	E 45	e 65	END O5	U 55	u 75	NAK 15
6 36	& 26	00	F 46	f 66	ACK 06	V 56	V 76	SYN 16
7 37	’ 27	5E	G 47	g 67	BEL O7	W 57	w 77	ETB 17
8 38	(28	[5B	H 48	h 68	BSP 08	X 58	x 78	CAN 18
9 39	) 29	] 5D	I 49	i 69	HT O9	Y 59	y 79	EM 19
: 3A	* 2A	00	J 4A	j 6A	LF CA	Z 5A	z 7A	SUM 1A
; 3B	+ 2B	00	K 4B	k 6B	VT OB			
, 2C	< 3C	{ 7B	L 4C	l 6C	FF 0C			
- 2D	= 3D	- 5F	M 4D	m 6D	CR 00			
. 2E	> 3E	} 7D	N 4E	n 6E	CO CE			
/ 2F	? 3F	\ 5C	O 4F	o 6F	CI OF			

FUNCTION KEYS

	NORM	SHFT	CTRL
BREAK	05	03	1B
ENTER	0D	0D	0D
SPACE	20	20	20
<-	08	18	10
->	09	19	11
v	0A	1A	12
^	0C	1C	13