microware

MICROWARE SYSTEMS CORPORATION
1900 N.W. 114th Street
Des Moines, Iowa 50322

Phone: 515-224-1929
Telex: 910-520-2535
FAX: 515-224-1352

The OS-9 Starter/Intermediate/Advanced Seminar

Please help us to improve our training seminars. Please complete this evaluation form and return it to the instructor or send it to Microware Systems, 1900 NW 114th St., Des Moines, IA 50325-7077, ATTN: Training and Education

Date of Course:_____
Instructor:_____

Please rate the quality of coverage on each topic:

		Excellent	Good	Adequate	Marginal	Inadequate
Starter (Monday)	Shell Overview	○	○	○	○	○
	Multi-User Overview	○	○	○	○	○
	Module Structure	○	○	○	○	○
	OS-9 C Language Basics	○	○	○	○	○
	OS-9 Program Debugging	○	○	○	○	○
Intermediate (Tues-Wed)	Module Overview	○	○	○	○	○
	Process Creation / Paths	○	○	○	○	○
	Pipes	○	○	○	○	○
	Data Modules	○	○	○	○	○
	Signals	○	○	○	○	○
	Alarms	○	○	○	○	○
	Events	○	○	○	○	○
	IPC Lab	○	○	○	○	○
Advanced (Thurs-Fri)	Process Scheduling	○	○	○	○	○
	Memory Management	○	○	○	○	○
	Subroutine Modules	○	○	○	○	○
	C / Assembly Interface	○	○	○	○	○
	Trap Handlers	○	○	○	○	○
	Interrupt Service Routines	○	○	○	○	○

microware

MICROWARE SYSTEMS CORPORATION
1900 N.W. 114th Street
Des Moines, Iowa 50322

Phone: 515-224-1929
Telex: 910-520-2535
FAX: 515-224-1352

What topic(s) do you feel were most beneficial? _____

What topics were not covered that should have been? _____

For the subjects covered, this course was: (too long) (the right length) (too short)

Would you have been able to attend a longer course? _____

Please write any general comments about the course that were not covered in the preceding questions.

TESTIMONIAL STATEMENT

"_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____"

"The statement above is my personal opinion of Microware's training and does not necessarily reflect my employer's opinion of Microware's training. Microware has my permission to use all or part of my testimonial in Microware's literature and/or advertising materials."

Signed: _____

Date: _____

Company: _____

Title: _____

Phone Number: _____

Thank you for your cooperation.

MICROWARE SYSTEMS CORPORATION

Corporate Headquarters • 1900 N.W. 114th Street • Des Moines, Iowa 50325-7077 • Phone: (515) 224-1929 • Fax: (515) 224-1352

Office of the
Training
and Education
Coordinator

Microware
Systems
Corporation
1900 N.W. 114th Street
Des Moines, Iowa
50325-7077
Tel: 515/224-1929
Fax: 515/224-1352

Name: _____

Company: _____

Mailing Address: _____

_____

_____

_____

Phone: ( ) _____

Fax: ( ) _____

E-Mail Address: _____

Course/Dates: _____/_____

Specific Questions:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

COPYRIGHT AND PUBLICATION INFORMATION

Publication date: June, 1994
Product Number: BND68NA68SL

DISCLAIMER

The information contained herein is believed to be accurate as of the date of publication. However, Microware will not be liable for any damages, including indirect or consequential, resulting from the use of this documentation. The information contained herein is subject to change without notice.

TRADEMARKS

OS-9, OS-9000 and Ultra C are trademarks of Microware Systems Corporation. All other product names referenced herein are either trademarks or registered trademarks of their respective owners.

Microware Systems Corporation • 1900 N.W. 114th Street
Des Moines, Iowa 50325-7077 • Phone: 515/224-1929

Training and Education
Presents:

OS-9 Resident Navigation

Welcome!

Welcome to this OS-9 training course! Before you may continue, you should log into the system this course is being taught on. Logging in requires a user-name and, perhaps, a password. Your instructor will provide you with your user-name and, if required, your password as well. All students begin the training session with a user-name in the form of **userX**, where **X** is a unique number for each student. During your training session, your user-name may change, but your assigned user number (**X**) will remain with you for the duration of your course(s) this week.

Logging Into The System

When user security is implemented on the OS-9 system, a program called **tsmon** monitors all ports that may be used to log in. Tsmon (time sharing monitor) waits for a carriage return to be entered on each port being monitored, and once the carriage return arrives, tsmon allows the **login** program to run on that port. Login will prompt for a user-name, and if applicable, a password for that user. If the user-name is recognized and the password is correct, login allows the user to access the system. This access is normally granted by allowing the user to run a program called **shell**. Shell is the standard command interpreter supplied with the OS-9 system. (Other command interpreters are also available for OS-9.)

The Shell Interface

Once you have logged into the resident OS-9 system you are given a prompt from a program whose job is to get your commands. This program, which may be called **shell**, will get your input and perform some task after processing it. Although your prompt differs, the prompt shown in this material will always be the dollar sign ($) since that is the default prompt.

In the eyes of the computer, the shell is nothing more than another program, and since OS-9 is a multi-tasking system, many people can run the shell at the same time.

Some basic commands

Here are several simple-to-use commands that can get you started. The **dir** command is used to find out the names of all files in a current directory. The command **pd** shows you the absolute pathlist to your current directory while **chd** allows you to change it. **Procs** is used to discover what processes you are currently running. None of these (or any other) commands are case sensitive unless they reside on a compact disk. (Normal OS-9 file systems are case insensitive, but the file system for a CD does rely on the case of files).

Command Options

Most commands accept input on the command line directing them to modify the way the command is to be executed. For example, the dir utility accepts the **-e** option to force the utility to display its output in extended format. Try the command *dir -e* and you will notice the difference. Procs also accepts the -e option, but the option has a very different meaning: procs' -e tells the utility to display information for every process on the system, rather than just the user's processes.

On-Line Help

If you would like to find out what options a command accepts, your first intuition should be to consult your manual. However, since that option will often follow "sleeping on it", Microware has written a small amount of on-line help into each of its utilities. If you choose the **-?** option for any Microware utility you will be presented with a list of what command options the particular command accepts on the input line. Try this (*dir -?*) with the dir utility to discover how the output will appear.

OS-9 File Structure

The OS-9 file structure is hierarchial in nature. That is, data is placed within directories, which may in turn also be within directories, creating a structure somewhat resembling a record tree. As previously mentioned, the chd command allows you to change your current directory. (Note, that since chd is not an actual command but is rather built into the shell, and as such there is no on-line help available through -?.)

The required syntax for chd is *chd <destination>*. The destination directory may either be specified as an absolute path or one that is relative to your current directory. (Recall, the pd command displays the name of your current directory.) If you have used other directory-based systems, you may be familiar with .. notation (pronounced dot-dot) as a shortcut referring to one directory level higher. OS-9 takes this notation a little further: you may add a third dot to refer to a second directory level up, a fourth dot, and so on. There is no limit to the number of dots that may be specified. You are protected, however, from going beyond the root directory of your current disk.

Disks on OS-9 are all separated from one another in the file system. Put another way, to change directory from one disk to another always requires an absolute path; relative paths just won't cut it. An absolute path is one beginning with a slash (/). The characters following the slash (and before a second slash) contain the name of the device representing the disk.

Environment Variables

Just as any program may have global variables, the OS-9 system supports a set of variables that programs may access, called environment variables. These variables are kept in an "environment" that is copied from one program to another when new processes are started. The environment of a process is kept in a private area, not accessible to any other processes, and may be of any size.

Environment variables may have any purpose programmed into individual applications. Some of the more common environment variables include: PORT, HOME, SHELL, USER, PROMPT, TERM, and _sh. Note, that case is sensitive in environment variables.

The command **printenv** is used to display your shell's environment. No arguments are accepted with printenv.

The command **setenv** is used to create or change an environment variable. It takes exactly two arguments: the name of the variable and the contents to assign to it.

The command **unsetenv** takes exactly one argument, the name of a variable, and removes it.

Environment Variables and the C Language

The simple program below demonstrates how to view the contents of an environment variable.

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   char *myname;
   myname = getenv("MYNAME");
   if (myname==NULL)
      printf("You must set MYNAME first!\n");
   else
      printf("Hello, %s\n",myname);
   exit(0);
}
```

This program can be input with the **build** utility which takes the name of the file to create as its only parameter, as in *build envestX.c*. (Remember to replace X with your user number!) Once the program has been entered, compile it with the command *cc envtestX.c*. The **cc** program, Microware's C compiler, will spend some time compiling and create a program called **envtestX**. Assuming no errors occur during compilation, type the command *envestX* to run your program.

Shell Command Line Modifiers

There are a number of special character sequences which may be entered on the command line to modify how the command will run. These sequences are not handled by the application but rather by the shell, prior to starting your application. These sequences are shown in Table 1.

Table 1: Shell Command Line Modifiers

Modifier	Description
>	redirection of standard output (stdout)
<	redirection of standard input (stdin)
>>	redirection of standard error (stderr)
!	pipe creation between processes
;	sequential command separation
&	concurrent command separation
^	priority of process
#	additional stack space operator
(,)	command grouping operators
*, ?	filename wild card characters

Background Operation

Using the concurrent command separator, commands can be made to execute in the background. By executing a command in the form *command &,* you are telling the shell to start up a process but do not wait for it to complete; merely issue another command prompt and await further input. Thus, running "in the background" means the process is in the background with respect to your shell.

The procs command is useful for discovering what process you have in the background.

Bringing processes to the foreground

Since a "background" process is that way only with respect to your shell, bringing a process to the foreground entails telling your shell to wait for it to complete. There is no way to tell the shell to wait for a particular process, but there are two commands that tell the shell to place itself in the background with respect to your other processes. The **w** command tells your shell to wait until one of its child processes has terminated before continuing. As soon as any of its children terminate, the shell will present another prompt for input. The **wait** command tells the shell to wait for all of its children to complete before issuing the next prompt.

Interrupt Characters

The OS-9 system supports two interrupt characters, or "hot keys," that, when pressed, get immediate attention. (When other characters are pressed, they enter a buffer of incoming data, but when a hot key is pressed, it is not entered into a buffer.) Though the hot keys can be reassigned, their default values are found in Table 2.

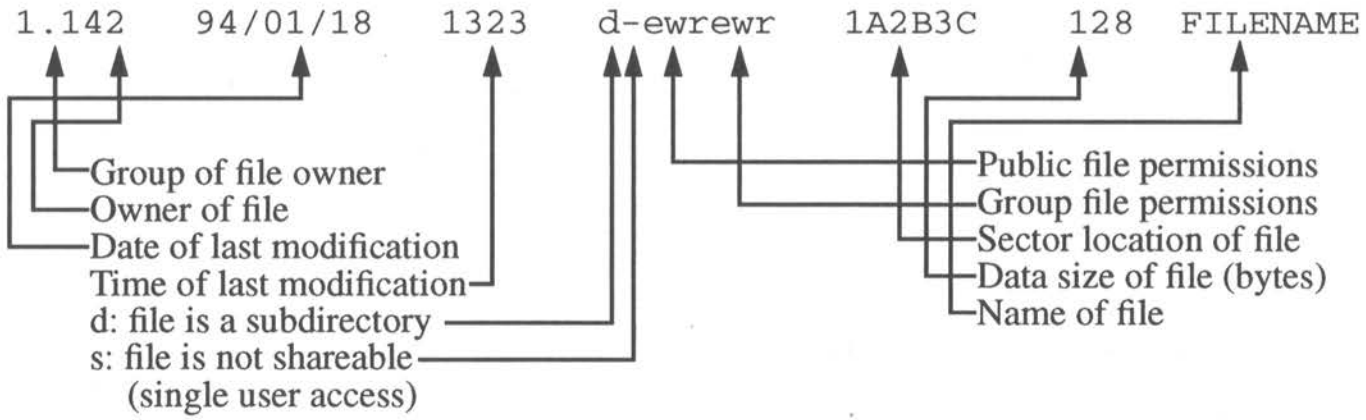
Table 2: Hot Keys

Key	Name
^C	SIGINT
^E	SIGQUIT

Each of these keys causes a signal (named in column two) to be sent to the last process to perform I/O on the device the keystroke came from. If that process is your command, the signal will possibly kill your process. If the last process to access the device is your shell, the action taken will depend on the signal itself. SIGINT causes the shell to stop waiting for children to terminate, while SIGQUIT will simply be passed along to the last process the shell created.

Extended Directory Listing

DIR is used to list the contents of a directory and is commonly used with the '-e' option to acquire an extended-information listing. This option produces, for each file in the directory being listed, a line of information similar to what follows.



Other Directory Related Commands

Besides dir, there are several other commands whose functions pertain to the file system in one way or another. We have already seen dir and mentioned chd and pd. Some additional commands include: **chx**, **makdir**, **deldir**, and **dcheck**. (Chd and chx are both built into the shell.)

pd: This command, seen earlier, shows your current data directory by default. It also accepts the '-x' option to display your current execution directory.

chx: Similar to chd, this command allows you to change your current execution directory.

makdir: This command allows you to create a new subdirectory.

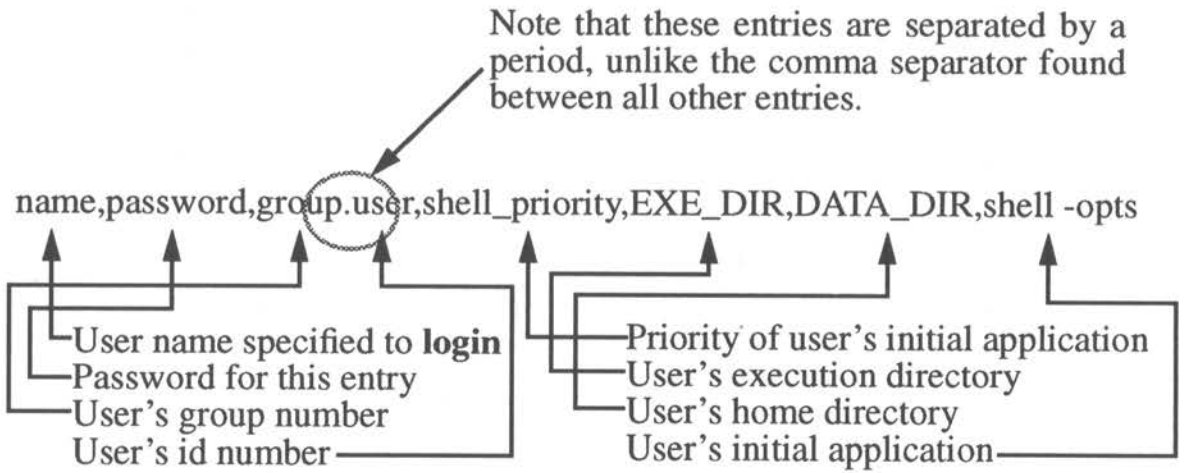
deldir: Allows you to remove a subdirectory and its contents.

dcheck: Checks the integrity of the file system and, optionally (-r) allows you to repair problems.

Multiple User Setup

In order for other users to log into the OS-9 system, they must have a valid entry in the file /dd/SYS/password. (Usually, this file cannot be read by users other than the super user. The Training and Education machine used in class is an exception.)

Each line in the password file appears as follows:



Text Editing

The text editor supported under OS-9 is called **umacs**, which is sometimes pronounced "micro-emacs" or "micro-macs" from the original spelling of macs. It is usually invoked with a command line such as *umacs <filename>* where <filename> is the name of a file to edit or create.

This editor is usually in "insert" mode, meaning, any text you type (other than special key combinations) will be inserted into the document. This editor is a very powerful tool, though it does require memorization of several non-intuitive commands to master its capabilities. Many of the more simple command sequences are listed below.

Table 3: Common Macs command sequences

Command	Description
^P	Move to the previous line
^N	Move to the next line
^B	Move backwards one character
^F	Move forewords one character

Table 3: Common Macs command sequences

Command	Description
^V	Move down a page
^Z	Move up a page
^A	Move to the beginning of the line
^E	Move to the end of the line
^D	Delete the character under the cursor
<BS>	(backspace key) Delete the character to the left of the cursor
^K	Delete (kill) to the end of the current line, place text into "kill buffer"
^Y	Yank text from kill buffer to current cursor position
<ESC>Z	Save your document and exit the editor
^X S	Save your document
^X ^C	Exit

More on the C Compiler

Using the macs editor , create the small program shown below (called mytestX.c) in the directory called C_DIR, which is a subdirectory from your home directory. Once the program has been written, save the file and compile it. (*cc mytestX.c*).

```
/*** mytestX.c ***/
main()
{
   get_term_defs();
   cls();
   cursor(0,0,"<firstname>");
   cursor(0,70,"<lastname>");
   cursor(12,35,"<middlename>");
   cursor(23,0,"");
   exit(0);
}
```

You should expect to see a few unresolved references when you compile. (Try *cc mytestX.c*) This is caused by making references to functions which are not part of the libraries we are including.

Multi-source Compiling

The example just compiled failed due to some unresolved function references. Specifically, functions get_term_defs(), cls(), and cursor() are not defined in the C libraries. (That covers everything you tried to do in that program!) The reason you were instructed to write this code within the C_DIR directory is because there is a file here containing the functions you are missing. To compile your program including this second source file, issue the command:

```
cc mytestX.c c_ctrl.c
```

This command will take a while to complete, but when it does, you will receive more unresolved reference errors! This time they are caused by c_ctrl.c making references to functions that are also not part of the standard library. These functions are, however, more or less standards; they are part of a library that is not automatically included, however.

Compiling with Additional Libraries

You need to make a small change to your compilation command, telling the C compiler to bring in a special library: "curses.l". If you reissue your command as:

```
cc mytestX.c c_ctrl.c -l=curses.l
```

your program should compile without any errors. Once the compilation completes, execute your program to make sure it works as you expect it to. What's that? Error 000:216 File Not Found? What did you type in? *mytestX*? Hmmm. You're pretty sure it should work; check out the files in your execution directory and see what *cc* created. Any new files? Just one, called output, huh? Well, go ahead and run it, lets see what happens.

```
output
```

Well, it seems to execute, but it won't be reliable.

Naming Your Executables

If you were able to get the correct information on screen when you entered the command *output*, you were lucky. If you change your current data directory to match your execution directory and then use the command *rename output mytestX*, the results may surprise you. The file will have a new name, but when you type the command *mytestX*, you may still find yourself running another executable. The reason for this will be made clear at the beginning of the Intermediate course. For now, you should compile your program one more time, specifying on the command line the name you want your final product to have:

```
cc mytestX.c c_ctrl.c -l=curses.l -f=mytestX -olM=5k
```

Note, that the executable needs more than the default 3k of stack space. The option on the end of the command line allocates 5k for this binary.

Compiling to Objects

To get that example to compile, we had to go through several iterations, most of which were really unnecessary and time consuming. If, rather than trying to go directly from '.c' files to an executable, we had compiled each source file to an object first, things would have gone considerably quicker.

Ultra-C supports two types of object, or intermediate files. The first is assembled code which needs to be linked with libraries to create an executable; this is the traditional object file, and has the extension of '.r'. The second, an "icode" file, is an Ultra-C specific object type. Compiling to icode takes longer but leaves more room for code optimization by the compiler. Thus, in the development setting you will probably want to perform traditional compiling and when your product is ready to ship, a final icode compilation may be in order.

To compile to an object you simply need to tell the compiler to stop early. The command option "-e<phase>" tells cc what phase to stop after. To stop at the assembled object level, stop at phase as, i.e., *cc -eas mytestX.c*. This creates the file "mytestX.r" which can be combined with c_ctrl.r and the termlib library to create our executable. The "io" phase is the icode optimizer, a good point to stop after if you are interested in an icode file called mytestX.i. Icode can be grouped along with object code and the required libraries to generate a final executable.

Project Management

There is a utility called **make** on the OS-9 system which has been designed to improve productivity in updating executables generated from (reasonably) large numbers of source files. (This utility is very similar to utilities of the same name found on other systems.) Make's function is to allow you to build your final product by separately compiling all items for your project and then linking them together, by typing one command. The utility reads a text file which basically tells make what you are building and how it gets done.

This text file, often called "makefile," contains macros, dependency lists, and rules. Macros are simply there to ease the job of maintaining dependencies and rules. Dependencies describe the relationship between high level items and low level items and rules tell make how to create a high level item. For the project we built minutes ago, the makefile shown below could suffice:

```
# lines beginning with a # are comments
ODIR = /h0/CMDS/exe       # our execution directory

mytestX: mytestX.r c_ctrl.r
   cc mytestX.r c_ctrl.r -l=curses.l -f=mytest
```

Debugging Under OS-9

Table 4: Debuggers Available for OS-9

	Debug	**SrcDbg**	**SysDbg**	**RomBug**
State	user	user/*system*	system	system
Language	asm	C/asm	asm	asm
Expressions	yes	YES	yes	yes
Break Points	yes	YES	yes	yes
Watch Points	no	yes	no	no

Microware is committed to providing more than technologically-advanced software.

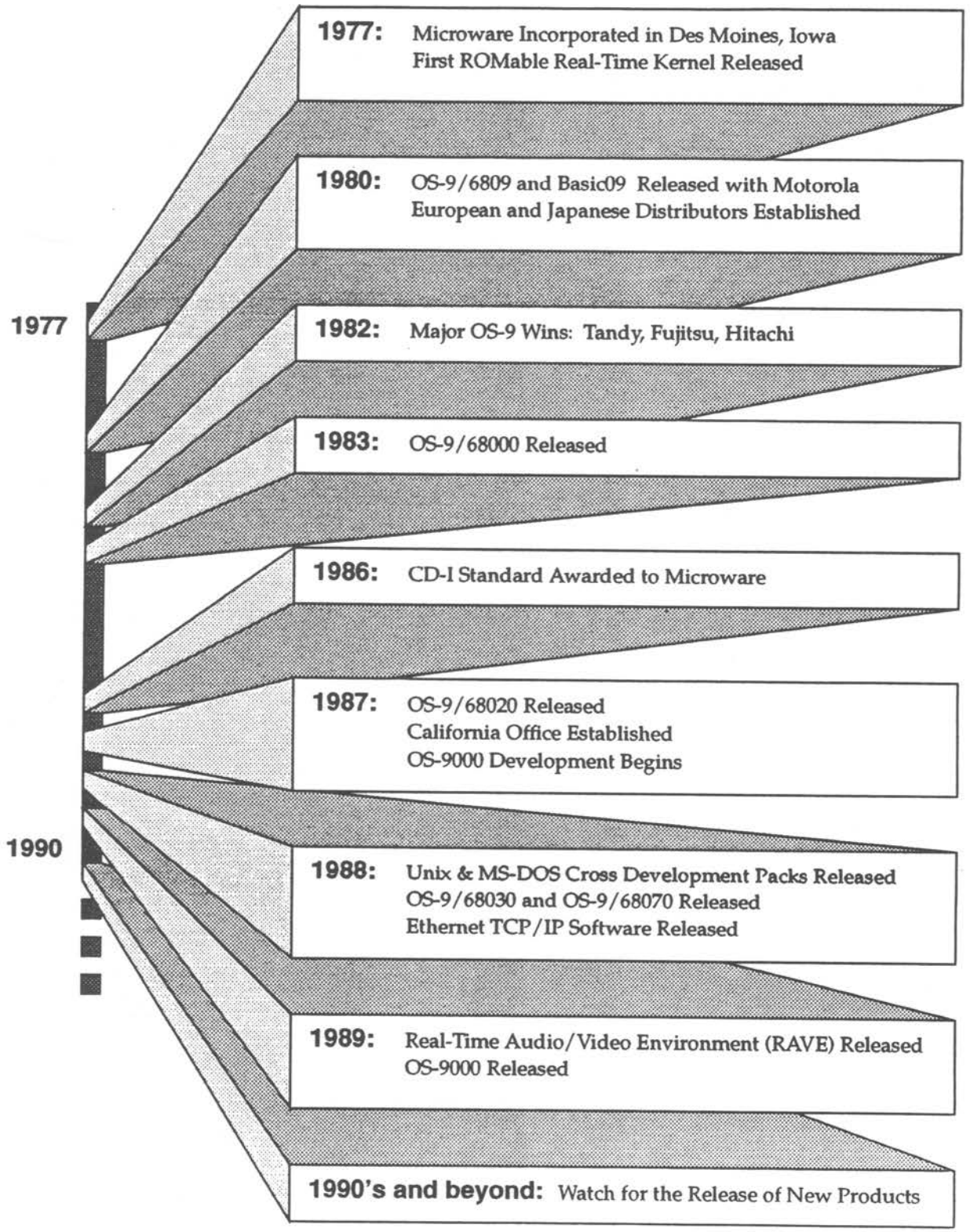
❖

Microware was conceived as a "total solution" software supplier, dedicated to serving its clients.

❖

Microware develops software products designed to address the needs of today with solutions for tomorrow.

Important Dates in Microware's History



OS-9 Version 2.4 Release Highlights

Complete details covering the Version 2.4 release of OS-9/680X0 may be found in the Release Notes, OEM Installation Manual and ROMbug Manual. OS-9/680X0 release highlights include:

- **68332 Support**
  OS-9/68332 includes full support for the MVME BCC Evaluation System and can be licensed in both Port Pak and distribution license configurations.
- **New ROM Debugger (ROMBug)**
  OS-9/680X0 now includes a symbolic ROM-based debugger. ROMBug incorporates many of the features of the System State Debugger and can be used to debug interrupt service routines.
- **Variable Sector Size RBF**
  RBF now supports variable sector sizes. This enhancement can significantly increase disk I/O throughput and media efficiency. Logical sector sizes can be any integral binary power ranging from 256 to 32768.
- **RBF Caching Support**
  RBF now supports write-through disk caching. This caching system increases disk I/O throughput by accessing structures in system memory and reducing external bus usage. A new `Diskcache` utility has been added to enable/disable cache, set cache size and report usage statistics.
- **Extended Non-Contiguous Boot Files**
  RBF boot files can now be non-contiguous and as large as any file allowed by a given device. This feature is especially attractive for systems which need to load large modules as a part of the bootstrap file.
- **SCSI Support for hard disks, floppies and tape drives**
  SCSI software includes support for the Common Command Set, connect/disconnect and a unique modular design that simplifies the installation process. SCSI software will be included in license updates and installation paks for Professional OS-9, RBF and SBF.
- **C Booting Technology**
  ROM boot-code can now be developed using C language technology to simplify sophisticated boot options on multiple devices.
- **A new Profile Shell command has been added**
  `Profile` allows the shell to receive input from a file and then return to the original input path.
- **Professional OS-9 and TapePaks now include a Tapegen utility**
  `Tapegen` allows the user to install bootstrap files on magnetic tape.
- **Industrial OS-9 now includes 13 new utilities:**
  `Deiniz, Dump, Echo, Exbin, Help, Ident, Iniz, Link, Printenv, Sleep, Tmode, Unlink` and `Xmode`.

Microware Systems Corporation
1900 N.W. 114th Street
Des Moines, Iowa 50325-7077

Telephone: 515 224-1929
FAX: 515 224-1352
Telex: 910 520-2535

OS-9 Topics - Course Syllabuses

Starter

1 Day Course

- The OS-9 User Interface

  Using the Shell
  - Environments
  - Redirection and modifiers

  Using Utilities
  - Execution directory
  - Directory layout (CMDS, USR, IO, DEFS, LIB, etc.)

  Making OS-9 Multi-User
  - Group/User numbers
  - Permissions
  - Creating password file entries

- Using the C Compiler

  Compiler options

  Compiling multiple source programs

  Using the Make utility

- Using the Debuggers

  Available debuggers

  The C source debugger - Srcdbg

Intermediate

2 Day Course

Day One

- The OS-9 Module Overview

  Memory module

  Kernel / Init

  Trap handlers

  I/O system

- Process Creation

  Process descriptor

  OS9exec

- Path Manipulation

  Path table

  I/O functions

- Interprocess Communication

  Pipes (named / unnamed)
  - Creating pipes
  - Using pipes
  - Error conditions

Microware Systems Corporation
1900 N.W. 114th Street
Des Moines, Iowa 50325-7077

Telephone: 515 224-1929
FAX: 515 224-1352
Telex: 910 520-2535

OS-9 Topics - Course Syllabuses, *continued*

Intermediate

Day Two

- Interprocess Communication, *(continued)*

Data Modules
- Shared RAM
- Creating and using data modules
- Module related functions

Signals/Alarms
- Sending and receiving
- Predefined signals
- System state signals

Events (semaphores)
- Creating events
- Waiting for a condition
- Signaling an event
- Example protocols
- Other calls

IPC Lab
- Create and play a simple and enjoyable game using:

  Pipes and signals

  or

  Data modules and events

Advanced

2 Day Course

Day One

- Process Scheduling
  - Process states
  - Moving between queues
  - Scheduling algorithm
  - Implementing real-time objectives
- Memory Usage
  - Global memory map
  - Memory allocation functions
  - Colored memory - installation and uses
- Subroutine Modules
  - Building a subroutine module
  - substart.a
  - Using a subroutine module
  - Rules and regulations
- Advanced C Topics
  - Assembly to C interface
  - Assembly generation from C compiler
  - Create your own C-binding in assembly

Advanced

Day Two

- Exception Handling

Boot Procedure
- Bootcode
- Kernel

Trap Handlers
- Run time library
- Installing
- Executing

Process (Program) Exception Handling

Customization Extension Modules
- Adding your own system calls to the kernel

Interrupt Service Routines
- Installing on a vector table
- Demonstration

The OS-9 Shell

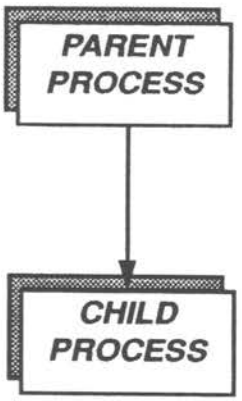
The shell is the OS-9 command interpreter program. The shell takes the commands you enter and translates them into commands the operating system understands and executes.

The shell also provides a user-configurable environment to personalize the way OS-9 works on your system. You can use the shell to change the shell prompt, send error messages to a file, or backup your disk before you log out.

Multiple Shells

Multiple processes can simultaneously use the shell utility. This allows each user to have their own shell. It also allows a user to have more than one shell.

Shells are easy to create. The quickest way to create a new shell is to type shell on the command line. The new shell is a descendent, or ***child process*** of the original, or ***parent shell***.



You can also create child shells by executing a ***procedure file***. A procedure file contains shell commands, one command per line. When the file is invoked, the shell ***forks*** (creates) a shell to execute the commands as if each command had been entered on the shell command line.

The child shell automatically accepts and executes the command lines from the procedure file instead of a terminal keyboard. This technique is sometimes called ***batch*** processing.

The Shell Environment

The shell maintains a list of ***environment*** variables for each user on an OS-9 system. These variables affect the operation of the shell or other programs subsequently executed and can be set according to the user's preference.

All environment variables can be accessed by any process called by the environment's shell or by descendant shells. This allows you to use environment variables as ***global*** variables.

If an environment variable is redefined by a subsequent shell, the variable is only redefined for that shell and its descendents. The environment is not redefined for the parent shell.

Four environment variables are automatically set up when you log on to a time-sharing system:

PORT — Specifies the name of the terminal. An example of a valid name is /t1. PORT is automatically set up by the tsmon utility.

HOME — Specifies your ***home*** directory. The home directory is specified in your password file entry and is your current data directory when you first log on the system.

SHELL — The name of the shell program to run.

USER — The user name you type when prompted by the login command.

For single user systems, you can set these variables with the setenv command. You can also set up a procedure file with your normal configuration of these variables. You could execute this file when you boot up your system.

There are four other important environment variables:

PATH — Specifies any number of directories to be searched when a command is executed.

PROMPT — Sets the current prompt.

_sh — Specifies the base level for counting the number of shell levels. _sh is the only standard environment variable in lower case letters.

TERM — Specifies the type of terminal being used.

Environment variables are case sensitive. OS-9 will not recognize a variable if the proper case is not used.

Changing the Shell Environment

Three utility programs are available for use with environment variables: printenv, setenv, and unsetenv.

- printenv prints the variables and their values to standard output. For example:

```
$ printenv
PATH=/dd/cmds:/h0/cmds:/d0/cmds
PORT=/term
HOME=/ho/usr/markd
SHELL=shell
USER=mark
PROMPT=$
_sh=0
```

- setenv declares the variable and sets its value. The variable is put in an environment storage area accessed by the shell. setenv requires two parameters. For example:

```
$ setenv PATH /dd/cmds:/dd/cmds/tel:/h0/cmds:/d0/cmds
$ setenv _sh 0
$ setenv SOURCE /h0/usr/tel/code
```

- unsetenv clears the value of the variable and removes it from storage. For example:

```
$ unsetenv PATH
$ unsetenv _sh
```

Command Line Processing

Execution modifiers, separators, and wildcards are not passed to the program as parameters when the shell processes a command line. Instead, these characters are stripped from the command line and processed by the shell. The following is a list of the available execution modifiers, separators, and wildcards:

modifiers:	#	Additional Memory Size
	^	Process Priority
	>	Redirect Output
	<	Redirect Input
	>>	Redirect Error Output
separators:	;	Sequential Execution
	&	Concurrent Execution
	!	Pipe Construction
wildcards:	*	Matches Any Character(s)
	?	Matches a Single Character
grouping	( )	Group shell commands together

Execution Modifiers

The shell processes execution modifiers before the program is run. If an error is detected in any of the modifiers, the run is aborted and the error reported.

Execute program with a new stack size modifier (#). This increases the default stack size used to execute a program. This increase in stack size can be assigned in 1K increments. If the specified stack size allocation is smaller than what is normally allocated, the modifier is ignored. The increase in stack size allocation only affects one command. The following are examples of the additional stack size modifier:

```
recurse #100                    Uses 100K of stack size for recurs.
update #50 new old              Uses 50K of stack size for update.
```

^ If you want a program to run at a higher priority, the process priority modifier (^) is used. By specifying a higher priority, a process is placed higher in the execution queue. Also, processes with higher priorities receive more CPU time. The following is an example of a process priority modifier:

```
$ format /d1 ^255          Specifies that format has an initial priority of 255.
```

< Redirects standard input. The standard input path normally passes data from a terminal's keyboard to a program.

> Redirects standard output. The standard output path normally passes output data from a program to a terminal's display.

>> Redirects standard error. By default, the standard error path uses the same device as the standard output path.

NOTES: Redirection modifiers can be used before and/or after the program's parameters, but each modifier can only be used once in a given command line. Redirection modifiers can be used together to cause more than one of the standard paths to be redirected.

+ - You can use the plus and hyphen characters (+ and -) with redirection modifiers. The >- modifier redirects output to a file. If the file already exists, the output overwrites it. The >+ modifier appends the output to the end of the file.

Spaces may not occur between redirection operators and the device or file path. The following are examples of redirection modifiers:

```
shell <>>>/t1                  Redirects all standard paths to t1.
dir >/d1/savelisting           Redirects output of dir to a file.
```

Command Separators

A single shell input line can include more than one command line. These command lines may be executed sequentially or concurrently.

; ***Sequential execution*** causes one program to complete its function and terminate before the next program is allowed to begin execution. Commands are sequentially executed by separating the command lines with a semicolon (;). After initiating a program, the shell waits until the program it created terminates. The command line prompt does not return until the program has finished. If an error is returned by any program, subsequent commands on the same line are not executed regardless of the -nx option. In all other regards, a semicolon (;) and a carriage return act as identical separators. For example:

```
list newfile;  dir >/p      Executes the list command and then the dir command.
```

& ***Concurrent execution*** allows several command lines to run simultaneously. Commands are concurrently executed by separating the command lines with an ampersand (&). This allows programs to run at the same time as other programs, including the shell. The shell does not wait to complete a process before processing the next command. Concurrent execution begins a ***background program***.

The number of programs that can run simultaneously is not fixed; it depends upon the amount of free memory in the system and the memory requirements of the specific programs.

By adding an ampersand (&) to the end of a command line, regardless of the type of execution specified, the shell returns the ID number of the background process, returns command to the keyboard, displays the $ prompt, and waits for a new command. This frees you from waiting for a process or sequence of processes to terminate. For example:

```
tsmon /t1 /t2&                          Runs tsmon in the background.
dir >/p1&  list file1      Executes the dir command in the background at the same
                                time the list command is running in the foreground.
```

! Pipelines are constructed with an exclamation point (!). Pipelines consist of two or more concurrent programs whose standard input and/or output paths connect to each other using pipes. For example:

```
procs -e ! grep procs          Pipes output of procs to the grep program.
list data ! qsort              Pipes output of list to the qsort program.
dir -rus ! grep filename       This command is equivalent to a find command.
```

Wildcard Matching

The shell accepts wildcards in the command line to identify file and directory names. The two recognized wildcard characters are the asterisk and the question mark (* and ?).

* An asterisk (*) matches any group of zero or more characters. The shell searches the current data directory or the directory given in a path for matching file names. For example:

```
list f*          Lists all files beginning with the letter f, including files such as f.txt.
dir ../*.backup  Lists the files in the parent directory that end with .backup.
```

NOTE: OS-9 does not require you to append an extension (such as .doc) on filenames. Therefore, use of the asterisk as a wild card character in OS-9 has different results than when used in MS-DOS. For example, in OS-9:

```
dir *.*   Lists only files in the current directory with a <filename>.<ext>. File
          names without a .<ext> are not included in the list.
dir *     Lists all files in the current directory, regardless of whether the file
          name contains a .<ext>.
```

? A question mark (?) matches any single character. The shell searches the current data directory or the directory specified in a path for matching file names. For example:

```
list f??   Lists three-character filenames beginning with f.
del form?  Deletes files beginning with form and having one additional character.
```

Wildcards may be used together. For example, the command list *.? lists any files that end in a period followed by any character, number, or special character, regardless of what comes before the period.

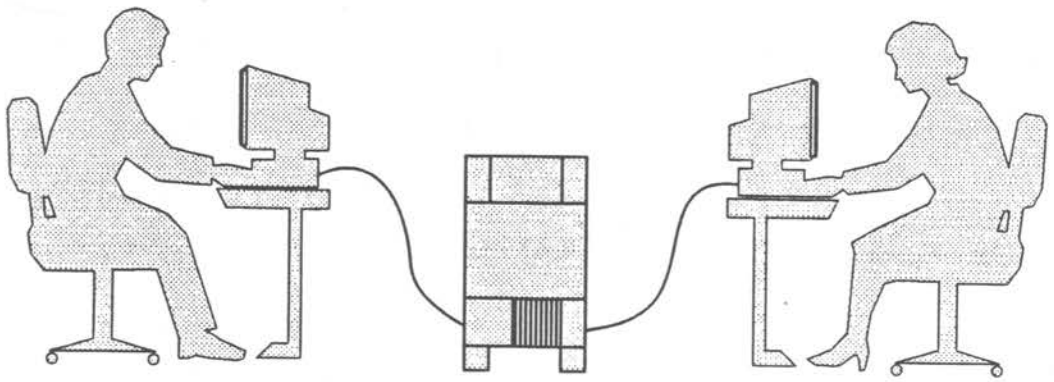
Command Grouping

( ) Sections of shell input lines can be enclosed in parentheses. This allows 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. It is important to remember that OS-9 processes commands from left to right. For example:

```
(dir /d0; dir /d1) >/p1          Directs output of both dir commands to /p1.
(del *.backup;procs -e >/p1)&    Runs del and procs in the background.
()                               With no commands, () forks a child shell.
```

Using OS-9's Multi-User Capabilities

OS-9 is a ***multi-user/multi-tasking*** operating system. This means that OS-9 supports the actions of more than one user at a time. Also, each user can run more than one process at a time.



Before adding users to your system, you should understand two concepts: ***file ownership*** and ***current directories***.

File Ownership

When a file or directory is created, a ***group.user ID*** is automatically stored with it. The group.user ID is formed from the user's group number and their user number. The group number allows people that work on the same project or work in the same department to be given a common group identification. The user number identifies a specific user. Therefore, a group.user ID identifies a specific user in a specific group or department.

The group.user ID determines file ownership. OS-9 users are divided into two classes: the owner and the public. The ***owner*** is any user with the same group or user number as the person who created the file. The ***public*** is any person with a group number that differs from the person who created the file.

A user with a group ID of 0 is referred to as a ***super user***. A super user can access and manipulate any file or directory on the system regardless of the file's ownership.

On multi-user systems, the system manager generally assigns a group.user ID for each user. This number is stored in a special file called a ***password file***. A super user on a multi-user system is generally the system manager, although other people such as group managers or project leaders may also be super users.

On single-user systems, users have super user status by default.

File Security System

File use and security are based on file attributes. Each file has eight attributes. The dir -e command displays these characters as an eight character string.

The term ***permission*** is used when one of the eight possible attribute characters is set. Permission determines who can access a file or directory and how it can be used. If a permission is not valid for the file or directory being examined, a hyphen (-) will be in its position.

Here is an attribute listing for a directory in which all permissions are valid:

```
dsewrewr
```

The attr utility changes file attributes. For example, the following command denies public read access of the file myfile:

```
attr -npr myfile
```

By convention, attributes are read from right to left. They are:

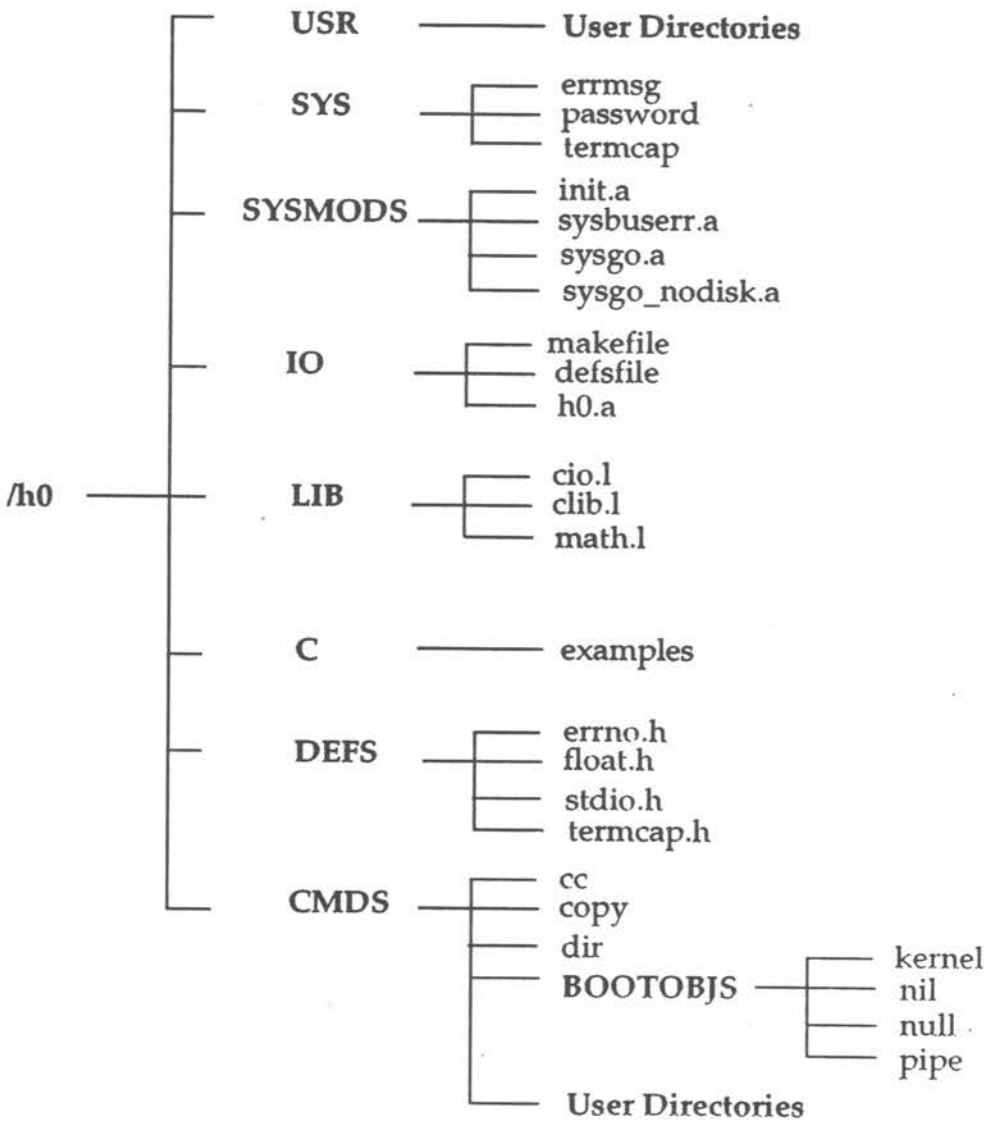
Attribute	Abbreviation	Description
Group Read	r	The group can read the file. When off, this denies any access to the file.
Group Write	w	The group can write to the file. When off, this attribute protects files from accidentally being modified or deleted.
Group Execute	e	The group can execute the file.
Public Read	pr	The public can read the file.
Public Write	pw	The public can write to the file.
Public Execute	pe	The public can execute the file.
Single user	s	When set, only one user at a time can open the file.
Directory	d	When set, indicates a directory.

Current Directories

Two working directories are always associated with each user or process. These directories are the ***current data directory*** and the ***current execution directory***. You create and store your text files in a data directory. Executable files, such as programs you have created and utilities, are located in an execution directory.

The current directory concept allows you to organize your files while keeping them separate from other users on the system. The word ***current*** is used because it is possible for you to move through the tree structure of the OS-9 file system to a different directory. This new directory would then become your current data or execution directory.

The following is an example of how a directory structure on a multi-user system might look:



The Password File

On multi-user systems, a password file should be located in the **SYS** directory. Each line in the password file is a login entry for a user. The line has seven fields separated by a comma:

① User name. The user name may contain up to 32 characters including spaces. If this field is empty, any name will match.

② Password. The password may contain a maximum of 32 characters including spaces. If this field is omitted, no password is required for the specified user.

③ Group.user ID number. Both the group and the user portion of this number may be from 0 to 255. This number is used by the file security system 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.

④ Initial process priority. This number may be from 1 to 65535. It indicates the priority of the initial process. By convention Microware suggests 128 as the default priority.

⑤ Initial execution directory. This field is usually set to /h0/CMDS or the user's execution directory.

⑥ Initial data directory. This is usually the specific user directory.

⑦ Initial program. This field contains the name and parameters of the program to be initially executed. This is usually the shell.

NOTE: Fields left empty are indicated by two consecutive commas.

The following is a sample password file:

```
general,,200.1,128,/h0/CMDS,/h0/USR,shell
superuser,secret,0.0,255,/h0/CMDS,/h0,shell -p="@howdy"
account,false,1.3,125,/h0/cmds/account,/h0/usr/account,shell
amy,mark,13.153,128,/h0/CMDS/AMY,/h0/USR/AMY,shell
justin,dragon,3.10,128,/h0/CMDS,/h0/USR/JUSTIN,Basic
```

Users may have more than one login entry. A single user name can have as many password file lines as desired, although the first line reached sequentially with a valid password is the password line used. For example, if user amy is also a super user, another line may be added to the password file:

```
amy,love,0.153,128,/h0/CMDS/AMY,/h0/USR/AMY,shell
```

If amy enters love at the password prompt, she will have super user privileges. She will have non-super user privileges if she enters mark at the password prompt.

Setting Up a Time-Sharing System Startup Procedure File

OS-9 systems used for time-sharing usually have a procedure file that brings the system up by means of one command or the system startup file. This procedure file initiates the time-sharing monitor for each terminal. It begins by starting the system clock and initiating concurrent execution of a number of processes that have their I/O redirected to each time-sharing terminal.

tsmon is a special program which monitors a terminal for activity. Typically, tsmon is executed as part of the start-up procedure when the system is first brought up and remains active until the system shuts down.

tsmon normally monitors I/O devices capable of bi-directional communication, such as CRT terminals.

It is possible to run several tsmon processes concurrently, each one watching a different group of devices. Because tsmon can monitor up to 28 device name pathlists, multiple tsmon processes must be run whenever more than 28 devices are to be monitored. Multiple tsmon processes are useful for other reasons. For example, it may be desirable to keep modems or terminals suspected of hardware trouble isolated from other devices in the system.

Here is a sample startup file for a time-sharing system with terminals named TERM, T1, T2, T3, and T71:

```
* system startup procedure file
echo Please Enter the Date and Time
setime </term
tsmon /t1 /t2 /t3&
tsmon /t71&
```

In this example, setime has its input redirected from the system console terminal. If the input was not redirected, OS-9 would attempt to read the time information from the current standard input path which is the procedure file instead of from the keyboard. Also notice that the ampersand (&) modifier is used to place tsmon in the background. This allows you to perform other tasks on your terminal while tsmon continues to run in the background.

NOTE: This startup file should not be used until a /h0/SYS/Password file with the appropriate entries has been created.

Setting Up a Multi-User System

Now that the basic principles have been discussed, the steps involved with making an OS-9 system multi-user are relatively simple:

① Decide and document the group and user numbers.

② Create the directories that the users will need. For each user, create a sub-directory in the USR directory and a sub-directory in the CMDS directory.

③ Edit the password file to allow each user to log in.

④ Copy/write a .login file for each user. Each time a user logs in, the shell will run the .login file just before issuing its first prompt. A typical .login file might look like the following:

```
-1
setenv TERM st
setenv PATH /h0/cmds:/n0/TrEd/h0/cmds
setenv EDITOR umacs
date
```

A .logout procedure file that runs when the user exits the login shell may also be created.

⑤ Make the startup file run tsmon on each port on the system.

At this point, your system should be up and running multiple users.

OS-9 Utility and Built-in Shell Command Summary

The following is a list of the utilities and built-in shell commands that come with the Professional OS-9 package. A ‡ preceding a description specifies a built-in shell command.

Utility	Description
attr	Display or change file attributes
backup	Make backup copy of a disk
binex	Convert binary data to S records
break	Invoke the system level debugger or reset the system
build	Build a short text file from standard input
cfp	Create/execute a temporary procedure file
chd	‡Change your current data directory
chx	‡Change your current execution directory
cmp	Compare two files
code	Return the hex value of a terminal key
compress	Compress an ASCII file
copy	Copy file(s)
count	Count characters, lines, and words in a file
date	Display the system date and time
dcheck	Check directory/file integrity
deiniz	Detach device(s)
del	Delete file(s)
deldir	Delete a directory
devs	Display the system's device table
dir	Display the contents of a directory
dsave	Copy a directory structure
diskcache	Enable, disable, or display status of cache
dump	Formatted display of file contents
echo	Echo text to output
edt	Line-oriented text editor
ex	‡Terminate execution of parent process and begin execution of child process
exbin	Convert S record(s) to binary data
expand	Expand a compressed file
fixmod	Fix module CRC and parity
format	Format an RBF device
free	Report free space on disk
frestore	Restore directory structure(s)
fsave	Backup a directory structure
grep	Search file for lines matching expression
help	Display usage of OS-9 utilities
ident	Display module information

Utility	Description
iniz	Attach devices
irqs	Display the system's IRQ polling table
kill	‡Abort specified process
link	Increment a memory module's link count
list	List contents of a file
load	Load a module into memory
login	Timesharing security log-in system; Execute .login file
logout	‡Terminate current shell; Execute .logout file
makdir	Create a new directory file
make	Program maintenance tool
maps	Reformat display of data memory
mdir	Display module directory
merge	Merge file(s) to standard output
mfree	Display system memory information
moded	Edit certain OS-9 modules
os9gen	Create boot on disk
pd	Print path of data or execution directory
pr	Display text file in specified format
printenv	Display shell environment variables
procs	Display user/system process information
profile	Read comments from file and return
qsort	Quick sort of text file
rename	Rename a file or directory
romsplit	Split a ROM image file
save	Save memory modules to files
set	‡Set shell options
setenv	‡Set environment variable to value
setime	Set system date and time
setpr	‡Set process priority
shell	Command interpreter
sleep	Suspend process for ticks/seconds
tape	Tape device special control commands
tapegen	Put files on a tape
tee	Copy input to multiple output paths
tmode	Display/change terminal characteristics on an SCF path
touch	Update the date of a file
tr	Convert all occurrences of characters in <string1> to <string2>
tsmon	Timesharing device monitor
unlink	Decrement a memory module's link count
unsetenv	‡Clear value of environment variable
w	‡Wait for last process to die
wait	‡Wait for all child processes to die
xmode	Display/change SCF device descriptor options

Microware's C Language Compiler System is a technologically advanced, high-performance, software development tool capable of the following:

- Comprehensive implementation of the full language
- Efficient code generation producing fast and compact object programs
- Generating position-independent, re-entrant, ROMable code
- High compilation speed
- UNIX and OS-9 compatible standard libraries

This compiler also serves as a gateway between UNIX and OS-9. Almost any application program written in C can be transported from a UNIX system to an OS-9 system, recompiled, and executed correctly. The compiler can also be run on UNIX-based computers, and the output can be downloaded to an OS-9-based 680x0 system.

Compiler Options

The compiler recognizes many command line options which modify the compilation process. Options are not case significant. All options are recognized before compilation begins. Consequently, the options may be placed anywhere on the command line. Options may be grouped together (such as -sr) except where an option specifies a parameter (such as -f=<path>).

Option	Description
-a	Suppresses the assembly phase, leaving the output as assembler code in a file with a .a suffix.
-bg	Sets the ***sticky*** bit in the module header. This causes the module to remain in memory even if the link count becomes zero.
-bp	Prints the parameters passed to each compiler phase and prints an exit status message.
-c	Outputs the source code as comments with the assembler code. This option is most useful with the -a option.
-d<identifier>	Equivalent to a #define <identifier> in the source file. Use this option when different versions of a program are maintained in one source file and differentiated through the #ifdef or #ifndef pre-processor directives. If <identifier> is used as a macro for expansion by the pre-processor, 1 is the expanded value unless an expansion string is specified using -d<identifier>=<string>.
-e=<number>	Sets the edition number constant byte to the specified number. This is an OS-9 convention for memory modules.
-f=<path>	Overrides the output file naming conventions. The last element of <path> specifies the name of the output file. The module name will be the same as the file name unless overridden by the -n=<name> option. The -f=<path> option causes an error if either the -a or -r options are also present. If <path> is a relative pathlist, it is relative to the current ***execution*** directory.
-fd=<path>	Similar to the -f=<path> option with the following exception: if <path> is a relative pathlist, it is relative to the current ***data*** directory.

Option	Description
-g	Causes the linker to output a symbol module for use by the symbolic assembly language level debugger. The symbol module has the same name as the output file with .stb appended. If a STB directory exists in the target output directory, the symbol module is placed there. Otherwise, it is placed in the same directory as the output file.
-i	Links the program with the cio.l library, causing references to selected C I/O functions to be handled by the cio trap handler module.
-j	Prevents the linker from creating a jump table.
-k=<n>[w|l][cw|cl][f]	<n> Specifies the target machine: 0=68000 (default), 2=68020. w Causes 16-bit data offsets to be generated. The default is 68000. l Causes 32-bit data offsets to be generated. The default is 68020. cw Causes 16-bit code references to be generated. The default is 68000. cl Causes 32-bit code references to be generated. The default is 68020. f Causes c68020 to generate 68881 instructions for float/double types.
-l=<path>	Specifies a library file to be searched by the linker before the standard library, math libraries, and system interface library.
-m=<mem size>	Instructs the linker to allocate <mem size> for the program stack. Specify the memory size in kilobytes. The default stack size is approximately 2K.
-n<name>	Specifies the output module's name.
-o	Inhibits the assembly code optimizer pass. The optimizer shortens object code by about 11% with a comparable increase in speed. It is recommended for production versions of debugged programs.
-q	Specifies quiet mode: the executive does not announce internal steps as they occur. Only error messages, if any, are displayed.

Option	Description
-r[=<dir>]	Suppresses linking library modules into executable programs. Output is left in files with a .r suffix. If -r=<dir>, .r files stay in <dir>.
-s	Stops the generation of stack-checking code. This option should only be used with great care when the application is extremely time critical and when the use of the stack by compiler generated code is fully understood.
-t=<dir>	Causes the executive to place the temporary files used by any compiler phase in the directory named <dir>. If the device containing the directory is a RAM disk device (for example, -t=/r0), compilation time will be drastically reduced.
-u<name>	Undefines previously defined pre-processor macro names. Macro names pre-defined in the pre-processor are OSK and mc68000. These names are useful for identifying the compiler under which the program is being compiled for the purposes of writing machine and operating system independent programs.
-v=<dir>	Specifies an additional directory to search for pre-processor #include files. File names within quotes are assumed to be in the current directory. The specified directory is searched for file names within angle brackets (<>). This option may appear more than once. If this option appears more than once, each directory is searched in the order given on the command line. The default DEFS directory is searched after all specified directories have been searched.
-w=<dir>	Specifies the directory containing the default library files (cstart.r, clib.l, etc.).
-x	Causes the compiler to generate trap instructions to access the floating point math routines. This option should appear on the command line when the program is both compiled and linked (if performed separately). The linker causes the object program to use the trap handler modules rather than extracting code from the math libraries.

Linker Command Line Options

The linker (l68) transforms the r68 assembler output into a single OS-9 format memory module. Many modules require more than the basic memory module requirements. The contents of the assembly language relocatable output files (ROFs) provide the information required to create each type of memory module. The linker allows references to occur between modules in order for one module to reference a symbol in another module. This involves adjusting the operand of many machine-language instructions.

When a program is being assembled, the assembler does not know the addresses of names which are external references to other program sections. Therefore, when an external reference is encountered, the assembler will set up information in the ROF which identifies the instructions that reference external names. Because the assembler is not aware of what the actual offset within the module is, each section is assembled as though it starts at offset 0.

The linker uses the ROFs produced by the assembler as input. The linker reads all the ROFs and then assigns each ROF a relative starting offset for its data storage space and a relative starting offset for its object code space.

The following options can appear on the command line. Options are case significant:

Option	Description
-a	Converts out-of-range bsr instructions and PC-relative lea instructions to jump table references. bsr instructions that address labels of 32K distant are automatically converted to jsr instructions using a jump table (in the initialized data area) that contains the desired destination address. lea instructions are changed to move instructions that move the destination from a jump instruction in the jump table. The linker automatically builds the required jump tables and includes them in the output file. This allows large programs to overcome the +/- 32K offset limit of bsr instructions without violating the OS-9 requirement for position independent code.
-e=<n>	Sets the module edition number. <n> is used for the edition number in the final output module. 1 is used if this option is not specified.

Option	Description
-g	Outputs symbol modules for use by the user and/or source debugger. If the .r files were created with the -g option, two symbol files are created: one file name with .stb appended and the other with .dbg appended. If not compiled with the -g option, only the .stb file is created. If a STB directory is present in the current execution directory, the symbol files are placed there. Otherwise, they are placed in the current execution directory.
-j	Prints jump table calculation map. See the description in the -a option.
-l=<path>	Uses <path> as a library file. A library file consists of one or more merged assembly ROF files. Each psect in the file is checked to see if it resolves any unresolved references. If so, the module is included in the final output module. Otherwise, it is skipped. No mainline psects are allowed in a library file. You can repeat this option up to 32 times in one command line to specify multiple library files. Library files are searched in the order given on the command line. The standard definition files are sys.l for assembly language or clib.l for the C compiler.
-M=<mem>[k]	Adds <mem>K to the stack memory allocation.
-m	Prints the linkage map indicating the base addresses of the psects in the final object module.
-n=<name>	Uses <name> as the module name.
-o=<path>	Writes linker object (memory module) output to the specified file, relative to the execution directory. The last element in <path> is used as the module name unless overridden by the -n option.
-O=<path>	Writes linker object (memory module) output to the specified file, relative to the data directory. The module name is the last element in <path> unless overridden by the -n option.
-p=<n>	Sets the permission word in the module header to <n>. <n> must be hexadecimal.
-r	Outputs a raw binary file for a non-OS-9 target system. The output will not be in memory module format.

Option	Description
-r=<n>	Outputs a raw binary file for non-OS-9 target systems with an object code base address at absolute address <n>. <n> must be a hexadecimal address. The base address is used to make absolute addressing references operate correctly.
-s	Prints a list of relative addresses assigned to symbols in the final object module. The symbols are listed in numeric order. This option is usually used with the -m option.
-S	Sets the sticky bit in the module header. This causes the module to remain in the module directory even if the link count becomes zero.
-w	When used with -s, this option displays symbols in alphabetic instead of numeric order.
-z	Reads module names from standard input.
-z=<file>	Reads module names from <file>.

COMMAND NAME	BINDING	DESCRIPTION
backward-character	^B	Moves cursor one character to the left
forward-character	^F	Moves cursor one character to the right
next-word	M-F	Moves cursor one word to the right
previous-word	M-B	Moves cursor one word to the left
next-line	^N	Moves cursor down the window one line
previous-line	^P	Moves cursor up the window one line
next-paragraph	M-N	Moves cursor ahead to the next paragraph
previous-paragraph	M-P	Moves cursor back to the last paragraph
next-page	^V	Moves cursor ahead one window
previous-page	^Z	Moves cursor back one window
begining-of-line	^A	Moves cursor to the beginning of the line
end-of-line	^E	Moves cursor to the end of the line
begining-of-file	M-<	Moves cursor to the beginning of the file
end-of-file	M->	Moves cursor to the end of the file
goto-line	M-G	Moves cursor to the specified line

COMMAND NAME	BINDING	DESCRIPTION
search-forward	M-S	Moves the cursor forward to the first occurence of the specified search string
search-reverse	M-R	Moves the cursor backward to the first occurence of the specified search string previous to the original cursor position
hunt-forward	<unbound>	Moves the cursor forward to the first occurence of the current search string
hunt-backward	<unbound>	Moves the cursor backward to the first occurence of the current search string previous to the original cursor position
replace-string	^R	Substitutes all occurrences of the specified search string with the specified replacement string
query-replace-string	M-^R	Prompts to substitutes each occurence of the specified search string with the specified replacement string

COMMAND NAME	BINDING	DESCRIPTION
set-mark	M-. M-<space>	Set marked bound of the region
exchange-point-and-mark	^X^X	Exchange marked bound of the region with the cursor position
copy-region	M-W	Copy the marked region to the kill buffer
kill-region	^W	Delete the marked region
case-region-upper	^X^U	Change all letters in the region to upper case
case-region-lower	^X^L	Change all letters in the region to lower case
yank	^Y	Paste the kill buffer at the cursor position

COMMAND NAME	BINDING	DESCRIPTION
split-current-window	^X2	duplicate current window in new window
next-window	^XO	Move cursor to next window
previous-window	^XP	Move cursor to previous window
move-window-up	^X^P	Scroll current window up one line
move-window-down	^X^N	Scroll current window down one line
scroll-next-up	M-^Z	Scroll next window up one page
scroll-next-down	M-^V	Scroll next window down one page
shrink-window	^X^Z	Decrease size of current window
grow-window	^XZ <ctrl>X^	Increase size of current window
delete-other-windows	^X1	Display only current window

COMMAND NAME	BINDING	DESCRIPTION
delete-next-character	^D	Deletes character under cursor
delete-previous-character	^H <bs> <del>	Deletes character before cursor
delete-next-word	M-D	Deletes from cursor to end of word
delete-previous-word	M-^H M-<bs>	Deletes word up to cursor
delete-blank-lines	^X^O	Deletes blank lines between text
kill-paragraph	M-^W	Deletes paragraph under cursor
kill-region	^W	Deletes marked region
kill-to-end-of-line	^K	Deletes line starting at cursor position
yank	^Y	Inserts kill buffer (last deleted item(s))

COMMAND NAME	BINDING	DESCRIPTION
insert-space	^C	Inserts space character before cursor
quote-character	M-Q	Allows control character to be printed
newline	^M	Has the same effect as a carriage return
open-line	^O	Inserts newline character after cursor
new-line-and-indent	^J <linefeed>	Inserts newline character and indents line equal to previous line
handle-tab	^I	Redefines and/or inserts tab character
insert-file	^X^I	Inserts file from directory before cursor

Macro Commands

^X(	begin-macro
^A	begining-of-line
^I	handle-tab
^N	next-line
^A	begining-of-line
^X)	end-macro

^ escape
M- control

COMMAND NAME	BINDING	DESCRIPTION
exit-emacs	^X^C	Exits after prompting to save changed files
quick-exit	M-Z	Exits after saving all changed files

COMMAND NAME	BINDING	DESCRIPTION
list-buffers	^X^B	Lists buffers to be used by μMACS
select-buffer	^XB	Selects buffer to be edited
next-buffer	^XX	Selects next buffer in buffer list to be edited
name-buffer	M-^N	Changes the name of the buffer currently being edited
buffer-position	^X=	Displays status line giving the current buffer position in relation to the entire file
delete buffer	^XK	Deletes the specified buffer
execute-buffer	<unbound>	Executes buffer as an μMACS procedure file

COMMAND NAME	BINDING	DESCRIPTION
insert-file	^X^I	Inserts file at current cursor position
read-file	^X^R	Reads file into current buffer (overwrites current text)
find-file	^X^F	Reads file into a new buffer
change-file-name	^XN	Names or Renames file in current buffer
save-file	^XS	Saves changed file
write-file	^X^W	Writes file to specified name
i-shell	^XC	Forks Shell; control remains in the Shell until an <esc> character is entered
shell-command	^X!	Forks Shell; executes specified command; control is then returned to μMACS

Many types of files are dependent on various other files for their creation. If the files that make up the final product are updated, the final product becomes out-of-date. The make utility is designed to automate the maintenance and re-creation of files that change over a period of time. By keeping track of modifications to program sources, make can determine the need to recompile, reassemble, and/or relink the files necessary to create an object file.

make maintains the files by using a special type of procedure file known as a ***makefile***. The makefile describes the relationship between the final product and the files that make up the final product. For the purpose of this discussion, the final product is referred to as the ***target file*** and the files that make up the target file is referred to as ***dependents***.

A makefile contains three types of entries: dependency, command, and comment.

Dependency Entry

A ***dependency entry*** specifies the relationship of a target file and the dependents used to build the target file. The entry has the following syntax:

```
<target>:[[<dependent>]<dependent>]
```

The list of files following the target file is known as the ***dependency list***. No limit is placed on the length of the dependency list, and any number of dependency entries can be listed in a makefile. A dependent in one entry may also be a target file in another entry. There is, however, only one main target file in each makefile. The main target file is usually specified in the first dependency entry in the makefile.

Command Entry

A *command entry* specifies the particular command that must be executed to update, if necessary, a particular target file. make updates a target file only if its dependents are newer than itself. If no instructions for update are provided, make attempts to create a command entry to perform the operation.

make recognizes a command entry by a line beginning with one or more spaces or tabs. Any legal OS-9 command line is acceptable. More than one command entry can be given for any dependency entry. Each command entry line is assumed to be complete unless it is continued from the previous command with a backslash (\). You should not intersperse comments with commands.

For example:

```
<target>:[[<file>]<file>]
   <OS-9 command line>
   <OS-9 command line>\
   <continued command line>
```

CAVEAT: Spaces may not follow the backslash (\).

Comment Entry

A *comment entry* consists of any line beginning with an asterisk (*). All characters following a pound sign (#) are also ignored as comments unless a digit immediately follows the pound sign. In this case, the pound sign is considered part of the command entry. All blank lines are ignored. For example:

```
<target>:[[<file>]<file>]

   * the following command will be executed if the dependent
   * files are newer than the target file
   <OS-9 command line>  # this is also a comment
```

To continue any entry, place a space followed by a backslash (\) at the end of the line to be continued. All entries longer than 256 characters must be continued on another line. All continuation lines must follow the rules for its entry type. For example, if a command line is continued on a second line, the second line must begin with a space or a tab:

```
FILE: aaa.r bbb.r ccc.r ddd.r eee.r \
fff.r ggg.r
            touch aaa.r bbb.r ccc.r \
            ddd.r eee.r fff.r ggg.r
```

NOTE: Spaces and tabs preceding non-command, continuation lines are ignored.

Running Make

To run the make utility, type make, followed by the name of the file(s) to be created and any options desired.

make processes the data three times.

During the first pass, make examines the makefile and sets up a table of dependencies. This table of dependencies stores the target file and the dependency files exactly as they are listed in the makefile. When make encounters a name on the left side of a colon, it first checks to see if it has encountered the name before. If it has, make connects the lists and continues.

After reading the makefile, make determines the target file on the list and makes a second pass through the dependency table. During this pass, make tries to resolve any existing ***implicit dependencies***. Implicit dependencies will be discussed.

make does a third pass through the list to get and compare the file dates. When make finds a file in a dependency list that is newer than its target file, it executes the specified command(s). If no command entry is specified, make generates a command based on the assumptions given in the next section. Because OS-9 only stores the time down to the closest minute, make remakes a file if its date matches one of its dependents.

When a command is executed, it is echoed to standard output. make normally stops if an error code is returned when a command line is executed.

To understand the relationship of the target file, its dependents, and the commands necessary to update the target file, the structure of the makefile must be carefully examined.

Implicit Definitions

Any time a command line is generated, make assumes that the target file is a program to be compiled. Therefore, if the target file is not a program to be compiled, any necessary command entries must be specified for each dependency list. make uses the following definitions and rules when forced to create a command line:

Object files	Files with no suffixes. An object file is made from a relocatable file and is linked when it needs to be made.
Relocatable files	Files appended by the suffix: .r. Relocatable files are made from source files and are assembled or compiled if they need to be made.
Source files	Files having one of the following suffixes: .a, .c, .f, or .p.
Default compiler	cc
Default assembler	r68
Default linker	cc
Default directory for all files	Current data directory (.)

NOTE: Only use the default linker with programs using cstart.

Macro Recognition

In addition to recognizing compilation rules and definitions, make recognizes certain macros. make recognizes a macro by the dollar sign ($) character in front of the name. If a macro name is longer than a single character, you must surround the entire name by parentheses. For example, $R refers to the macro R, $(PFLAGS) refers to the macro PFLAGS, $(B) and $B refer to the macro B, and $BR refers to macro B followed by the character R.

Macros may be placed in the makefile for convenience or on the command line for flexibility. Macros are allowed in the form of <macro name> = <expansion>. The expansion is substituted for the macro name whenever the macro name appears.

A *macro definition* is a line containing a macro name, followed by an equal sign (=). Both trailing and leading spaces and tabs are ignored. The macro expansion may contain other previously defined macros. Typically, macros appear before the main text of the makefile, but they may appear anywhere in the makefile. You must define macros before they are used.

In order to determine what the proper command lines are, make uses *default rules*. The general forms of the default command lines are:

Compiler	`cc $(CFLAGS) $*.c -r=$(RDIR)`
Assembler	`r68 $(RFLAGS) $*.a -o=$(RDIR)/$@`
Linker	`cc $(LFLAGS) $(RDIR)/$*.r -f=$@`

By substituting the proper macro definitions into place, make builds the proper command for each file.

If all of the files needed to be made were found, make would generate the following output:

```
cc -t=/dd test1.c -r=rels
cc -t=/dd test2.c -r=rels
cc -t=/dd test3.c -r=rels
r68 -q test4.a -o=rels/test4.r
chd rels; cc -gi test1.r test2.r test3.r test4.r -f=test
```

If you define a macro in your makefile and then redefine it on the command line, the command line definition will override the definition in the makefile. This feature is useful for compiling with special options.

To increase make's flexibility, you can define special macros in the makefile. make uses these macros when assumptions must be made in generating command lines or when searching for unspecified files. For example, if no source file is specified for program.r, make searches either the directory specified by SDIR or the current data directory for program.a (or .c, .p, .f).

make recognizes the following special macros:

Macro	Definition
ODIR=<path>	make searches the directory specified by <path> for all files with no suffix or relative pathlist. If ODIR is not defined in the makefile, make searches the current directory by default.
SDIR=<path>	make searches the directory specified by <path> for all source files not specified by a full pathlist. If SDIR is not defined in the makefile, make searches the current directory by default.
RDIR=<path>	make searches the directory specified by <path> for all relocatable files not specified by a full pathlist. If RDIR is not defined, make searches the current directory by default.
CFLAGS=<opts>	These compiler options are used in any necessary compiler command lines.
RFLAGS=<opts>	These assembler options are used in any necessary assembler command lines.
LFLAGS=<opts>	These linker options are used in any necessary linker command lines.
CC=<comp>	make uses this compiler when generating command lines. The default is cc.
RC=<asm>	make uses this assembler when generating command lines. The default is r68.
LC=<link>	make uses this linker when generating command lines. The default is cc.

Some reserved macros are expanded when a command line associated with a particular file dependency is forked. These macros may only be used on a command line. When you need to be explicit about a command line but have a target program with several dependencies, these macros can be useful. In practice, they are wildcards with the following meanings:

Macro	Definition
$@	Expands to the file name made by the command.
$*	Expands to the prefix of the file to be made.
$?	Expands to the list of files that were found to be newer than the target on a given dependency line.

Creating Your Own Default Rules

You can also create your own default rules. For example, if you always want files ending in .txt to be processed a special way, you could build your own default rule:

```
.txt.set:
   spl $*.txt
   touch $*.set
```

NOTE: Any line beginning in the left most margin ends a default rule declaration.

Now in your makefile, you can add the following line:

```
myfile.set: myfile.txt
```

myfile.set is dependent on myfile.txt. If myfile.txt has been updated more recently than myfile.set when the make utility is used on this file, the file is spooled to the printer and the date is updated with the touch utility.

Existing default rules can also be changed in this way. For example, if you want to use a different option for C files, you could change the existing default rules:

```
.c.r:
   cc $(CFLAGS) -ix $*.c -r=$(RDIR)
```

Notice that the -i and -x options have been added. By default, all programs will now use the cio module and the math trap handler.

Make Generated Command Lines

make is capable of generating three types of command lines: compiler command lines, assembler command lines, and linker command lines.

① Compiler command lines are generated if a source file with a suffix of .c needs to be recompiled. The compiler command line generated by make has the following syntax:

```
$(CC) $(CFLAGS) $(SDIR)/<file>[.c] -r=$(RDIR)
```

② Assembler command lines are generated when an assembly language source file needs to be re-assembled. The assembler command line generated by make has the following syntax:

```
$(RC) $(RFLAGS) $(SDIR)/<file>.a -o=$(RDIR)/<file>.r
```

③ Linker command lines are generated if an object file needs to be relinked in order to re-make the program module. The linker command line generated by make has the following syntax:

```
$(LC) $(LFLAGS) $(RDIR)/<file>.r -f=$(ODIR)/<file>
```

WARNING: When make is generating a command line for the linker, it looks at its list and uses the first relocatable file that it finds, but only the first one. For example:

```
prog: x.r y.r z.r
```

The previous command line would generate cc x.r instead of cc x.r y.r z.r or cc prog.r.

Make Options

Several options allow make even greater versatility for maintaining files/ modules. These options may be included on the command line when you run make or they may be included in the makefile for convenience.

When a command is executed, it is echoed to standard output, unless the -s, or silent, option is used or the command line starts with an "at" sign (@). When the -n option is used, the command is echoed to standard output but not actually executed. This is useful when building your original makefile.

make normally stops if an error code is returned when a command line is executed. Errors are ignored if the -i option is used or if a command line begins with a hyphen.

Sometimes, it is helpful to see the file dependencies and the dates associated with each of the files in the list. The -d option turns on the make debugger and gives a complete listing of the macro definitions, a listing of the files as it checks the dependency list, and all the file modification dates. If it cannot find a file to examine its date, make assumes a date of -1/00/00 00:00, indicating the necessity to update the file.

If you want to update the date on a file, but do not want to remake it, use the -t option. make merely opens the file for update and then closes it, thus making the date current.

If you are quite explicit about your makefile dependencies and do not want make to assume anything, use the -b option to turn off the built-in rules governing implicit file dependencies.

The following is a list of the available options:

Options	Description
-?	Displays the usage of make.
-b	Does not use built-in rules.
-bo	Does not use built-in rules for object files.
-d	Prints the dates of the files in the makefile (Debug mode).
-dd	Double debug mode. Very verbose.
-f-	Reads the makefile from standard input.
-f=<path>	Specifies <path> as the makefile. If <path> is specified as a hyphen (-), make commands are read from standard input.
-i	Ignores errors.
-n	Does not execute commands, but does display them.
-s	Silent Mode: executes commands without echo.
-t	Updates the dates without executing commands.
-u	Does the make regardless of the dates on files.
-x	Uses the cross-compiler/assembler.
-z	Reads a list of make targets from standard input.
-z=<path>	Reads a list of make targets from <path>.

Putting It All Together

The following example is a makefile to create make:

```
* beginning
ODIR = /h0/cmds
RDIR = rels
CFILES = domake.c doname.c dodate.c domac.c
RFILES = domake.r doname.r dodate.r
PFLAGS = -p64 -nh1
R2 = ../test/domac.r
RFLAGS = -q
make: $(RFILES) $(R2) getfd.r
$(RFILES): defs.h
$(R2): defs.h
    cc $*.c -r=../test
print.file: $(CFILES)
    pr $? $(PFLAGS) >/p1
    touch print.file
*end
```

The makefile in this example looks for the .r files listed in RFILES in the directory specified by RDIR: rels. The only exception is ../test/domac.r, which has a complete pathlist specified.

Even though getfd.r does not have any explicit dependents, its dependency on getfd.a is still checked. The source files are all found in the current directory.

Notice that this makefile can also be used to make listings. By typing make print.file on the command line, make expands the macro $? to include all of the files updated since the last time print.file was updated. If you keep a dummy file called print.file in your directory, make only prints out the newly made files. If no print.file exists, all files are printed.

The OS-9 Debuggers

Before discussing OS-9's debuggers, two terms need to be defined: system-state and user-state processes.

System-state processes are processes running in a protected mode. System-state processes basically have unlimited access to system memory and other resources. When a process in system state wants to use the CPU, it waits until it has the highest age. Once it receives CPU time, a process in system state runs until it finishes, instead of running for a specified time-slice. System state is sometimes referred to as ***supervisor state***.

User-state processes do not have access to all points in memory and do not have access to all commands. When a process in user state gains time in the CPU, it will run only for one time-slice. When the process finishes it's time-slice, it will be entered back in the waiting queue according to its initial priority.

Four debuggers are available for OS-9:

- Debug
- Sysdbg
- SrcDbg
- Rombug

Differences Between the Debuggers

These debuggers differ in the state and language they use for debugging, the expressions that are allowed, whether breakpoints can be set, whether watchpoints can be set, and whether symbolic constants can be used. The following chart summarizes these differences:

	Debug	Sysdbg	SrcDbg	RomBug
state	user	system	user	system
language	assembly	assembly	C/assembly	assembly
expressions	simple	simple	any valid C expression	simple
breakpoints	yes	yes	yes‡	yes
watchpoints	no	no	yes	no
symbolic	yes	yes	yes‡	yes

‡ While all of the debuggers allow you to set breakpoints and use symbolic constants, SrcDbg allows the most freedom and flexibility in these areas.

Debug

Debug debugs and tests user-state 68000 machine language programs written for the OS-9 operating system. Debug uses software techniques to control the process being debugged. Debug uses the F$DFork and F$DExec system calls to create and execute the process to be debugged. These system calls provide an environment that allows the debugger to control the execution of a process without affecting other processes on the system. Full access to the 68000 user-mode registers is also provided.

Debug allows you to disassemble, trace, set breakpoints, and display data while programs are run. Because Debug runs in user state, it will not interfere with other users on the system. This means that code, such as drivers, that run in system state cannot be debugged with Debug.

Debug has symbolic capabilities. When debugging a program, Debug is aware of your symbols. If the program had the label Start: in it, di Start could be entered and Debug would know what address to begin the disassembly. An expression, even expressions involving symbols, may be used anywhere Debug requires a parameter.

Invoking Debug

Debug is invoked by typing debug at the shell prompt. Debug waits for a command with the prompt:

```
dbg:
```

The f command should be the first command entered in a Debug session. This creates a process to execute the program to debug. If any parameters are specified on the shell command line, they are assumed to be parameters for the f command. If any parameters are specified, the f command is implicitly executed upon startup. If redirection, priority specification, and stack size are given, enclose the parameters in quotes to protect them from interpretation by the shell. The f command can pass up to 64 arguments to the process.

The following is a summary of most of the commands that can be used with the user-state debuggers.

Relocation Registers

Debug maintains eight relocation registers. These registers are useful for sorting memory base addresses for later use in commands and expressions. The registers are referenced by the names r0 through r7. The r0 register is hardwired to zero. Whenever an address is specified, the default relocation register is added to the address automatically. Setting the default relocation register to zero disables this action. The default relocation register is not added if a symbolic address or an expression is specified.

The following commands deal with relocation registers.

Command	Description
@	Prints the default relocation register.
@<num>	Sets the default relocation register.
.r	Displays the relocation registers.
.r<num> <val>	Sets the register r<num> to <val>.

Breakpoints

Debug allows you to set ***breakpoints***. Breakpoints are positions in the code that tell the debugger when to stop and dump the contents of the registers and allow you to enter commands. The debugger sets up to 16 simultaneous breakpoint addresses. You can only set breakpoints at even-byte addresses. The debugger supports both soft and hard breakpoints. Each is used differently.

A ***soft breakpoint*** is not actually placed in the code, but is emulated by the F$DExec system call. This allows you to set breakpoints in ROM code or code that another process is currently executing. Because the soft breakpoint facility is implemented in the software, the program runs much slower than normal in this mode.

A ***hard breakpoint*** is an illegal instruction placed in the code to cause an illegal instruction exception. Because of this, you cannot use a hard breakpoint in ROM code. If another process is executing the code being debugged, it will most likely exit with an illegal instruction error when the breakpoint instruction is reached. The program runs at full speed when using hard breakpoints.

Command	Description
b	Displays the list of currently set breakpoints.
b <addr>	Sets a breakpoint at <addr>.
k <addr>	Kills the breakpoint at <addr>.
k *	Kills all breakpoints.

Execution Commands

Debug provides commands to control the execution of the debugged process. These commands are important for examining the flow of control and effects on data structures.

Command	Description
i	Displays a count of the executed instructions.
g	Runs the program from the current pc until the program terminates or hits a breakpoint.
g <addr>	Starts running from <addr> with the same stopping conditions as g.
gs	Used to "step" across bsr instructions, the debugger will run the subroutine and dump the registers upon return.
gs <addr>	Runs the program until <addr> is reached.
t	Causes the debugger to execute one instruction and dump the registers.
t <count>	Traces and dumps registers for <count> instructions.
x <count>	Executes <count> instructions without register dumps. x - 1 is used to debug a process at full speed with hardware breakpoints.

Memory Change Commands

To examine or change memory, use the debugger memory change command. When this command is used, the debugger automatically enters the *memory change mode*. The basic interface is the same: the debugger displays the current value of the location and you have a number of options (leave alone, change, go back one location, etc.).

Command	Description
c <addr>	Changes the memory in byte register increments.
cw <addr>	Changes memory by word.
cl <addr>	Changes memory by long.
.<reg> <val>	Changes a machine register.
mf <s> <e> <v>	Fills memory from <s> to <e> with value <v>.
ms <s> <e> <v>	Searches memory from <s> to <e> for value <v>.

Display Commands

Memory is displayed using the memory display command: d. This command allows interpretation of memory in a number of ways. The following are the common forms:

Form	Description
d <addr>	Displays memory from <addr> for 256 bytes.
di <addr>	Disassembles memory from <addr> for 16 instructions.
.	Displays a register dump.

Symbol Commands

The symbolic debugging facility of the debugger allows easy debugging without linkage maps or address tables for reference. The -g option of the linker (l68) writes symbol information into an OS-9 data module. The linker places the symbols associated with global code and data offsets into the symbol table. If a symbol module is available for the code module being debugged, you can use symbolic addresses in most debugger commands.

The following commands deal with the symbol table features:

Command	Description
s	Displays all symbols in all symbol modules.
sm	Displays symbol module table.
ss	Sets current symbol module to the module containing the current program counter.
ss <addr>	Sets current symbol module to the module containing <addr>.
ss <name>	Sets current symbol module to the symbol module specified by <name>.
sd <name>	Displays all data symbols for the specified symbol module.
sc <name>	Displays all code symbols for the specified symbol module.

Sysdbg

Sysdbg is a system state debugger intended for debugging both system-state and user-state programs. Sysdbg runs in system state and takes effective control of the MPU (Memory Process Unit) by causing the kernel to ignore other processes while it is active.

Sysdbg uses the normal I/O system calls to perform I/O to the controlling terminal and to load symbol modules from the disk. This technique allows Sysdbg to run on any OS-9 host system without customization of I/O routines. Because of this, Sysdbg cannot debug the serial I/O driver that it is using for terminal I/O or any IRQ service routines that service interrupts at IRQ levels greater than that of the terminal I/O hardware.

Sysdbg can be used only on a fully booted system because it is invoked with the F$Fork system call. The system-level bootstrap ROM debugger is normally used to debug a system until it is able to completely bootstrap from ROM or disk.

Sysdbg takes control of the system when active. Consequently, normal time-sharing activities are suspended until control is returned to the kernel. Sysdbg normally runs at IRQ mask level 0. Therefore, all device interrupts (both clock and non-clock) are serviced in the usual fashion. Breakpoints or tracing in IRQ service routines should be performed with care because interrupts below the mask level are not serviced. Avoid tracing or breakpointing in the portions of the kernel dealing with task scheduling and process exception handling because the system may loop or crash if Sysdbg executes any of this code on behalf of itself. System calls used implicitly by Sysdbg are F$Sleep, F$AProc, and F$NProc.

Sysdbg is usually run with no other users on the system because it can abruptly halt multi-tasking activities for long periods of time. For this reason, only super users can initiate Sysdbg.

Invoking Sysdbg

To invoke Sysdbg, type sysdbg at the shell prompt. There are no meaningful options or arguments.

Sysdbg performs a number of checks and validations upon entry. First, Sysdbg verifies that it was invoked by a super user. Non-super users are not allowed to run Sysdbg. Next, Sysdbg ensures that it is running on the proper version of the kernel. Sysdbg also verifies that the kernel's internal data structure formats are the same as what it expects. This prevents strange behavior on later releases of the kernel. Finally, Sysdbg verifies the MPU type (as determined by the kernel) as 68000, 68010, or 68020.

Sysdbg then enters the command mode. The first commands are usually a commands to attach to all the symbol modules corresponding to the code modules to debug. Breakpoints are then set at the appropriate addresses, and the g command is given to return to normal timesharing. When the breakpoint is reached, Sysdbg regains control.

The majority of Sysdbg's commands are the same as those for Debug. The following command is used in Sysdbg, but does not exist in Debug.

> A <mod> parameter attaches a module. This command specifies the module to debug because no program is specified at the initiation of the debugger.

NOTE: Sysdbg has no x command. Instead, the g command is used to run the process at full speed until breakpoints are hit. For more complete information on the System State Debugger, refer to the ***OS-9 System State Debugger User's Manual.***

SrcDbg

SrcDbg runs in user state like Debug and can debug both assembly and C language programs. You have control over the execution of the program and the values stored in variables. The command line features a C interpreter that can analyze C expressions. Full symbolic information is available at all times.

Invoking SrcDbg

The following is the syntax for the SrcDbg command line:

```
srcdbg {<scrdbg_opts>} [<prog>] {<prog_opts>} [["]<redirections>["]]
```

<srcdbg_opts> are the SrcDbg command line options:

-d	SrcDbg does not read the .dbg file.
-m[=]<memory>	Extra memory is allocated for <prog>.
-s	SrcDbg does not read the .stb file.
-z[=]<path>	SrcDbg reads commands from <path>.

<prog> is the name of the C program being debugged. <prog_opts> are passed directly to <prog> as command line arguments.

The following are frequently used commands. For a complete reference, see ***OS-9 Source Level Debugger User Manual***.

Program Execution Control Commands

fo[rk] <program> [<opts>]

Fo[rk] begins execution of the specified program. Once forked, the debug process' program counter points to the primary module's execution entry point. Any <opts> appearing on the fo[rk] command line are passed to the program as parameters. If no parameters are specified, fo[rk] re-starts the program with the same parameters used by either the last fo[rk] command or the SrcDbg command line.

g[o] [<location>]

G[o] executes the program starting at the current location. G[o] executes the program until a breakpoint is encountered, an exception occurs, a signal occurs, or the end of the program is reached.

s[tep] [<num>]

S[tep] executes the specified number of executable statements of the program. If no number is specified, S[tep] executes a single statement. SrcDbg executes a statement, displays the values of any changed watch expressions, and displays the next executable line.

n[ext] [<num>]

N[ext] executes the specified number of executable statements of the program. If a number is not specified, SrcDbg executes one statement. SrcDbg executes a statement, displays the value of any changed watch expressions, and displays the next executable line.

r[eturn] [<num>]

R[eturn] executes the program until the current function returns to the calling function or a breakpoint is encountered. If <num> is specified, r[eturn] executes until it returns to the specified number of callers above the current stack frame.

b[reak] [<location>] [:wh[en] <expr>] [:co[unt] <num>]

B[reak] sets a breakpoint at a specified line number, result of an <expr>, or upon a when <expr> becoming true. Breakpoints are generally used to stop execution in a program to allow single-line-stepping through specific areas. If no parameters are specified, all current breakpoints in the program are displayed.

w[atch] [<expr>]

W[atch] monitors the value of the specified <expr> as the program is executed. Each watch expression is evaluated each time a machine instruction is executed. The watch expression is displayed with its initial value when it is first set. SrcDbg then displays each watch expression only when its value changes.

k[ill] [<break>/<watch>] {,[<break>/<watch>]}

K[ill] removes all specified breakpoints (<break>) and watchpoints (<watch>). SrcDbg notation (i.e., b1, b2, w1, w2, etc.) specifies <break> and <watch>. You can use wildcards with this command.

l[o]g <path> | : off

L[o]g writes SrcDbg commands to <path>. The specified pathlist is relative to your current data directory. ": off" closes the log file.

re[ad] [<path>]

Re[ad] reads SrcDbg commands from <path>. The <path> is relative to your current data directory. The l[o]g command may be used to create the file referred to in <path>.

o[ption] {<opts>}

O[ption] allows you to set a variety of display and execution options.

l[ist] <param>

L[ist] with no parameters displays the next 21 lines of source code beginning with the current line number. If the end of file occurs before 21 lines are displayed, an end-of-file message is displayed. <param> may be any OS-9 pathlist, any source file known to SrcDbg, a scope expression resulting in a block number, a scope expression resulting in a function, or a line number expression.

l[nfo] [<data item/structure>]

l[nfo] returns information about specified program objects and the current program location. If no parameters are specified, SrcDbg displays the current location of the program.

f[rame] [<num>]

F[rame] displays stack frame information. If no parameters are specified, the name of each calling function, the location from which it was called, and the frame number are displayed. The active frame is designated with an asterisk (*).

p[rint] [<expr>]

P[rint] returns the value of the specified <expr>. P[rint] displays the value according to the resulting data type of the expression. All program objects referred to within the <expr> are relative to the current scope, with one exception: "static" variables outside the current scope may be accessed using scope expressions.

a[ssign] [<expr>] = [<expr>]

A[ssign] sets the value of a program object. The left side must resolve to a variable, but the right may be any C expression that can be assigned to the variable on the left.

con[text] [<data item>]

Con[text] displays the complete scope expression of an object. Frequently, con[text] is used to determine if you are accessing a local or global object.

fi[nd] <name>

Fi[nd] displays all scope expressions for <name> found in the debug process. If no parameters are specified, the previous fi[nd] command is repeated. If there was no previous fi[nd] command, an error message is returned.

lo[cals]

Lo[cals] displays the value of all the local variables.

asm(.)

Asm or "." displays the current register values and the current machine instruction.

c[hange][w | l |word | long] [<addr>]

C[hange] changes memory starting at the result of <addr>. When the c[hange] command is invoked, SrcDbg displays the first byte/word/longword at the location specified by <addr> and then waits for you to respond. If a C expression is entered at this point, the memory at that location is changed to the result of the entered expression.

dil[ist] <location> [: <num>]

Dil[ist] displays the current C source line and the assembly code which maps to the current C source line starting at the address specified by <location>. If dil[ist] is entered without any parameters, the previous dil[ist] command is executed. If there was no previous dil[ist] command, an error message is returned.

di[sasm] [<addr>] [: <count>]

Di[sasm] disassembles memory starting at the address specified by <addr>. If <count> is specified, <count> machine lines are disassembled and displayed. If <count> is not specified, 16 machine lines are disassembled and displayed. If di[sasm] is entered without any parameters, the previous di[sasm] command is repeated. If there was no previous di[sasm] command, an error message is returned.

d[ump] [<addr>] [<count>]

D[ump] returns a formatted display of the physical contents starting at the address specified by <addr>. If <count> is specified, <count> lines of information are displayed. If <count> is not specified, 16 lines of information are displayed.

t[race] [<num>]

T[race] executes the specified number of machine instructions. If <num> is not specified, t[race] executes a single instruction.

gostop(gs)

Gostop executes the specified number of machine instructions in the current subroutine.

Rombug

Rombug is a system-state ROM debugger for debugging both system-state and user-state programs. Rombug runs in system state and takes control of the MPU. The debugger's command set allows you to analyze programs by setting breakpoints, tracing control, and trapping exceptions. This can all be performed symbolically. Extensive memory commands allow you to examine and change register values.

Using the talk-through and download commands, you can also communicate with the host system as a terminal and download programs into RAM for testing via the communications link.

The debugger accepts command lines from the console that consist of a command code followed by a return key. You can use the backspace (control-H) and line delete (control-X) keys to correct errors.

Symbolic Debugging

The OS-9 linker (l68) produces a symbol module for a program if the l68 -g option is specified when the program is linked. This option causes global data and code symbols to be placed in a data module. The symbol table data module (STB) must be loaded in memory when the **a** (attach) command is executed. The program module's CRC is stored in the STB module and validated by Rombug to ensure that the symbol module matches the version of the code module being debugged.

Invoking Rombug

Rombug is called by one of the following methods:

- The ROM bootstrap
- The F$SysDbg system call
- The break utility
- The kernel calls Rombug in "system crash" conditions

Rombug can also be activated by any processor exception which it is monitoring. For example, a hardware abort switch which causes an exception could invoke Rombug. The other ways in which Rombug may be activated are discussed in the ***OS-9 ROM Debugger User's Manual***.

The first commands are usually commands to attach to all the symbol modules corresponding to the code modules to debug. Breakpoints are then set at the appropriate addresses and the g command is given to return to normal timesharing. When the breakpoint is reached, control is returned to Rombug.

Rombug's commands are similar to Debug. See the ***OS-9 ROM Debugger User's Manual*** for more information about the commands.

```
/*************************************************************
*  srcdem<n>.c                                               *
*************************************************************
*  Program to demonstrate how to use the source-level debugger *
*  called srcdbg.                                            *
*************************************************************/
@_sysedit: equ 10

#include <stdio.h>

struct st1 {
  char a;
  int b;
  double c;
} var1;

double func();
void recfunc();
/*******************  main *********************************/
main()
{
  double d;

  var1.a = 'R';
  var1.b = 10101010;
  var1.c = 3.14159265;
  d = func(var1);
  printf("d = %.81f\n",d);
  recfunc(0);
  exit(0);
}
/*******************  func  ********************************/
double func(par)
struct st1 par;
{
  double ret;

  ret = (double)par.a + (double)par.b + par.c;
  return (ret);
}
/*******************  recfunc  *****************************/
void recfunc(val)
register int val;
{
  if (val < 10)
      recfunc(val + 1);
  return;
}
/*******************  add    *******************************/
add(x,y)
int x,y;
{
  int temp1;

  temp1 = x;
  x = y;
  y = temp1;

    printf("%d + %d = %d\n", y, x, y+x);

    return(y+x);
}
/*************************************************************/
```