

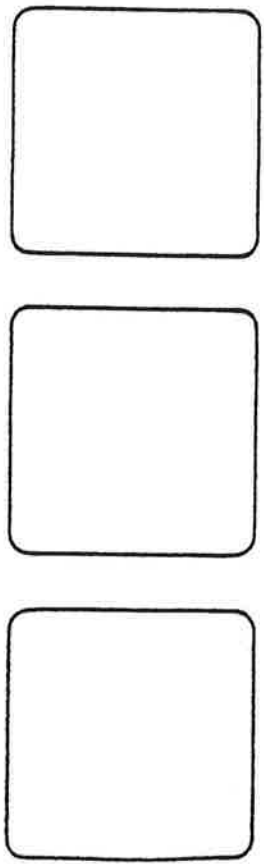
---

# COHERENT™

---

OPERATING SYSTEM

---



**C**ommodore

VOLUME 2



Mark Williams

## Table of Contents

|   |    |
|---|----|
| 1. Introduction                         | 1  |
| Hardware requirements                   | 2  |
| Software requirements                   | 2  |
| 2. Installing your COHERENT file system | 3  |
| Booting COHERENT                        | 3  |
| Building the COHERENT System            | 3  |
| Building the filesystems                | 4  |
| Creating additional file systems        | 5  |
| What went wrong?                        | 6  |
| 3. Beginning to use the COHERENT system | 7  |
| Start-up                                | 7  |
| Checking the file system                | 7  |
| Setting the date                        | 7  |
| Changing the time zone                  | 8  |
| Going to multi-user mode                | 8  |
| Going to multi-user mode                | 9  |
| 4. Tips on using the system             | 10 |
| Enabling cron                           | 10 |
| Adding new users                        | 10 |
| Using floppy disks                      | 10 |
| Copying floppy disks                    | 11 |
| Backing up your hard disk               | 11 |
| Normal shutdown                         | 12 |
| Named pipes                             | 12 |
| Differences from other documentation    | 13 |
| Possible problems                       | 13 |
| Touring the file system                 | 14 |
| 5. C900 Coherent 0.7.3 files            | 17 |
| System disk 1                           | 17 |
| System disk 2                           | 19 |
| System disk 3                           | 21 |
| 6. C900 COHERENT Manual Pages           | 24 |
| craa                                    | 24 |
| cu                                      | 25 |
| kermalt                                 | 27 |
|   | 29 |

## COHERENT Version 0.7.3 Preliminary Release Notes

June 19, 1985

Copyright (C) 1982-1985

Mark Williams Company  
 1430 West Wrightwood Avenue  
 Chicago, Illinois 60614  
 Telephone: (312) 472-6659

This document conveys information that is the property of Mark Williams Company. It shall not be copied, reproduced or duplicated in whole or in part without the express written permission of Mark Williams Company. COHERENT is a trademark of Mark Williams Company.

Mark Williams Company makes no warranty of any kind with respect to this material and disclaims any implied warranties of merchantability or fitness for any particular purpose.

The information contained herein is subject to change without notice.

Printed in U.S.A.

## 1. Introduction

Congratulations on purchasing the Mark Williams Company COHERENT (tm) operating system for the Commodore C900 personal computer. In addition to this document, your installation kit includes the following items.

- Three floppy disks, labeled *COHERENT 0.7.3 Volume 1* through *J*. These disks contain the root file system of your COHERENT system.

One of these floppy disks is labeled *COHERENT 0.7.3 Boot/build*. This is a boot disk configured specifically for the prototype C900.

- A set of COHERENT documentation, including:

*Introduction to COHERENT Documentation*  
*Introduction to the COHERENT System*  
*COHERENT Command Manual*  
*COHERENT System Manual*  
*COHERENT Administrator's Guide*  
*COHERENT Assembler Reference Manual*  
*awk User's Manual*  
*bc Calculator Language Tutorial*  
*ed Interactive Editor Tutorial*  
*learn User's Guide*  
*lex Lexical Generator Tutorial*  
*m4 Macro Processor User's Manual*  
*truff/ Text Processor Tutorial*  
*sh Shell Command Language Tutorial*  
*yacc Parser Generator Tutorial*

The *Introduction to the COHERENT System* is the manual you should read first. It is a tutorial, with simple, step-by-step examples. The *COHERENT Command Manual* describes each command available on the COHERENT system.

The procedure described in the next section of these notes will install the COHERENT system onto your hard disk. Please take a few minutes to read all the instructions before you begin the installation.

The remaining sections of these notes describe normal operating procedures for your COHERENT system; give device-specific information about your COHERENT system; and provide additional manual pages for the C900 COHERENT documentation.

As a user of the COHERENT system, you are in an especially important position to tell us how the system can be improved and whether any problems appear. We welcome your comments, and we know you will enjoy using the COHERENT system.

## Hardware requirements

To run the COHERENT system, you need the following:

1. A Commodore C900.
2. 512Kb of memory.
3. One double-sided double-track floppy disk drive.
4. A hard disk with 20 megabyte capacity.

## Software requirements

The COHERENT installation kit includes three floppy disks, which contain all of the commands and files you need to run the COHERENT system. These disks are labelled *COHERENT 0.7.3 Volume 1* through *J*.

## 2. Installing your COHERENT file system

The following instructions tell you how to install your COHERENT system on your C900. They tell you how to start the boot procedure and the build procedure from a floppy disk; how to set up additional file systems; and how to deal with problems encountered during the boot and build routines.

### Booting COHERENT

The term *boot* is short for *bootstrap*; it means running a simple program that in turn knows how to find the COHERENT system, set it up, and start it running -- in other words, the COHERENT system can pull itself up by its bootstraps.

To boot the COHERENT system, use the following procedure.

1. Your C900 comes with one double-sided double-track floppy disk drive, called drive 1. Insert the boot/build disk (volume 1), label side up, into drive 1 and close the door.

If the C900 is on, press the reset button found on the back of the computer's cabinet. Otherwise, turn it on. In either case, you will see on your console the message

```
Commodore 900 diagnostics
```

followed by a series of messages from the program that tests whether your C900 is working properly.

The diagnostics program will first attempt to boot COHERENT from the floppy disk. If the boot program cannot be found on the floppy disk, then the diagnostics program will try to boot COHERENT from the hard disk. If neither can be found, then it will access the diagnostic monitor and wait for you give it an instruction. To boot COHERENT from the diagnostic monitor, type

```
(rd,1)coherent
```

This tells the diagnostics program to look on the floppy disk drive again for the boot program.

### Building the COHERENT System

*The following steps destroy all information on the root partition. hdd, of your hard disk. If there is information on the hard disk in this partition which you wish to keep, copy it to floppy disks before you continue.*

*Building* means installing the COHERENT system onto your hard disk either for the first time or to install an upgraded version of COHERENT. If you happen to be re-installing the COHERENT system, be sure that all users are logged off; then execute the sync routine to ensure that no data are damaged during the re-building process.

To begin the build, insert the boot/build diskette into drive 1 and press the reset key on the back of the computer.

Soon you will receive the message

```
Coherent (484K, 7452) Version 0.7.3
(c) 1982 - 1985 Mark Williams Company, Chicago
#
```

The numbers in parentheses may differ on your system. Type:

```
build
```

The select light on the floppy disk drive will come on again as the system begins the build procedure.

### Building the filesystems

The build procedure will now warn you that

```
The next step will destroy all data on the
hard disk root partition. If you need to
save files, type ^ctrl-C to stop this procedure.
If you are ready to build the root file
system, type RETURN.
```

If you have nothing to save on the root partition, type <RETURN>.

The build program will then check your hard disk for bad blocks. While it is checking, the message

```
Checking the hard disk for bad blocks ...
```

will appear on your screen. You may get several error messages of the form

```
(2,0): dev: read error, block=number
status=19
```

These result from tracks marked bad when the hard disk was initialized at the factory. The purpose of this step is to insure that all bad blocks are properly noted by the COHERENT file system, to ensure that no data are written onto them.

After the root partition has been checked, you will receive the message

```
Writing a filesystem to the hard disk ...
```

and a few minutes later you will receive the message:

The hard disk system will now boot and load commands and utilities onto the root partition. Press the reset key, and be sure there is no floppy disk in the drive, so that I will boot from the hard disk.

Remove the floppy disk from drive 1 and press the reset button on the back of the computer.

After the computer reboots from hard disk, you will see:

```
Coherent (915k, 1101a) Version 0.7.3
(c) 1982-1985 Mark Williams Company, Chicago
```

Now, build will ask you for the two remaining floppy disks in order, and return the 'w' prompt after it has read them all. Reading the two floppy disks will take about 30 minutes.

#### Creating additional file systems

The build procedure will create only the root file system on its partition. You must build a file system on each of the additional partitions by using the `mkfs` command, as follows.

First, create a *proto* file called `pt`. Assuming that you want to use the partition `/dev/wd3` with 10,336 blocks in it, enter the following:

```
ed pt
a
/dev/null
10336 1400
wq
```

The first entry is a *boot block*. Use `/dev/null` because this is not a bootable file system. Normally, this is the file name of the boot block code. 10336 is the number of blocks on the device, and 1400 is the number of i-nodes, or files. As a rule of thumb, the number of i-nodes should be about one-seventh the number of blocks. Thus, if you have a partition with 5,184 blocks on it, 648 is an acceptable number for the i-nodes. The number of i-nodes is the largest number of files and directories that can be created on the partition.

If you have placed a swap device on a partition that is shared with space for files, subtract the size of the swap area from the partition size, and use the difference as the number of blocks. This will insure that blocks in the swap area are not used in the file system and thus overwritten by the swapper.

Now, enter the command

```
/etc/mkproto /dev/hd1 sp1 y2
```

This tells the system to read each block on the partition and maintain a list of blocks that caused read errors. These bad blocks are recorded by the file system, and will

never be used.

Enter the command

```
/etc/mkfs /dev/hd1 p2
```

to create the file system on the partition.

Repeat this process for each of the partitions.

#### What went wrong?

If your installation of COHERENT does not succeed, either your hardware is not configured correctly or the software supplied on floppy disks is unreadable or inappropriate for your system. Reread the installation instructions carefully, then repeat the installation procedure. If you still cannot install COHERENT, check the manuals that came with your C900 and follow their suggestions to insure that the hardware is operating correctly.

## 3. Beginning to use the COHERENT system

Once the COHERENT system is installed on your machine, you must boot COHERENT whenever you begin to work with the computer.

## Start-up

First, make sure that no floppy disk is in the floppy disk drive. Press the reset button on the back of the computer cabinet. The diagnostic/boot ROM will check the hardware, then it will say:

```
coherent (1915x, 1016) version 0.7.3
(c) 1982-1985 Mark Williams Company, Chicago
```

on the console screen. (The pair of numbers in the parentheses will vary with the amount of memory available on your machine.) On booting, the system is in single-user mode, and commands are executed with the permissions of the superuser, or root.

Whenever you reboot COHERENT, you should both check the file systems on your hard disk and set the date, as shown in the following sections.

## Checking the file system

To protect your files and the COHERENT commands from potential loss or damage, you should check the file systems on your hard disk whenever you reboot the system. You have four file system partitions: /dev/hd0, /dev/hd1, /dev/hd2, and /dev/hd3, with root device being /dev/hd0. The command

```
check -s /dev/hd3 /dev/hd2 /dev/hd1 /dev/hd0
```

will tell the system to check the consistency of the your files. It will let you know if anything is out of order, and attempt to reconcile any problems it finds (this is the meaning of the "-s" flag). The following procedure must be followed whenever check is invoked:

1. If you are checking the root device, list it last on the command line.
2. If errors are found on the root device, immediately reboot the system *without* typing sync. When the system is running again in single-user mode, re-issue the check command for each partition that was found to be in error, listing the root device last on the command line.
3. If there are errors on non-root partitions but not on the root partition, you may re-issue the check command without re-booting, listing each device that had errors.
4. If the errors persist after the second check, the file system must be repaired by hand, using the other file system utilities in command directory /bin. These are described in the *COHERENT Administrator's Guide*. You need to check only those partitions that contain files.

Situation (2) is the only instance when you should reboot without typing sync. Normally, the sync command must be issued to ensure that the file system is up to date before you reboot the system.

If the root file system is severely damaged, you may need to re-install the COHERENT system from the distribution kit.

For more information about repairing file systems, see the *COHERENT Administrator's Guide*.

## Setting the date

Whenever you boot the COHERENT system, you should reset the current date and time. The system notes the date and time whenever you create or modify a file. If the date and time settings are not accurate, some commands, e.g., make and at, will not function properly. If you type

```
date
```

the system will show you what it believes the current date and time to be. If you have not set the time zone parameter to your local time zone, the time will be displayed in Greenwich Mean Time, which is six hours ahead of Central Standard Time (Chicago). The section *Changing the time zone*, below, describes how to change the time zone.

If you have already set the time zone, set the date and time with the date command. Type:

```
date 8510271317:56
```

to set the date to October 27, 1985, and the time to 1:17:56 P.M.

## Changing the time zone

If you are not in the Greenwich time zone, you should change the file /etc/profile to specify your time zone. (/etc/profile is a file that is executed whenever a user logs on.) Do this with the timezone command. To use timezone, you must give the names of your standard and daylight-savings time zones, and the location of your time zone in minutes west of Greenwich, England. For example, the command

```
TIMEZONE=EST:300:EDT
```

tells the COHERENT system that your standard time zone is Eastern Standard Time (EDT), your daylight-savings time zone is Eastern Daylight Time (EDT), and that you are located 300 minutes west of Greenwich.

Note that the term *minutes west* refers to minutes of time, not minutes of longitude. To calculate how many minutes of time your city is west of Greenwich, multiply its degrees of longitude by four, for example, Chicago is located 88 degrees west, which means that it is 352 minutes (5 hours and 52 minutes) west of Greenwich *in time*. If you are unsure of the longitude of your city, check any atlas.

See the documents *Introduction to the COHERENT System* and *sh Shell Command Language Tutorial* for further information.

### Using floppy disks

Because of the flexibility of the COHERENT file system, you can build and use directories directly on floppy disks. For best use of floppy disks, you should understand the following procedures.

To build a file system on a floppy disk in drive 1, insert a formatted diskette into the drive and type the command

```
/etc/mkfs /dev/rd1 2300
```

The number 2300 is the number of blocks on the floppy disk file system.

The next step is to mount a floppy disk that contains the file system so you can access it. For example, type

```
/etc/mount /dev/rd1 /f0
```

to mount the floppy disk in drive 1 on directory /f0; then you can refer to files on the floppy disk as being in directory /f0. The directory /f0 has been provided on the system for you to mount floppy disks. The mount command will verify that there is a mountable file system on the disk. mount will also disallow mounts of more than one device on the same directory. mount will check that the mount table /etc/mntab is newer than the file /etc/boottime. /etc/boottime is a dummy file which serves simply to keep the time the system was last booted. If /etc/mntab is older than /etc/boottime, the mount table is possibly invalid.

The last step is to unmount a mounted floppy disk when you are finished using it. Type

```
/etc/umount /dev/rd1
```

to unmount the floppy disk in drive 1.

Because the COHERENT system buffers all data it writes to or receives from a floppy disk, if you do not unmount your floppy disk you may lose data from this disk or from the next disk you put in the drive.

### Copying floppy disks

You can use the dd command to copy the floppy disks in your COHERENT distribution kit or to copy other floppy disks. Be sure to copy your boot floppy disk and keep the original in a safe place with the other floppy disks from the kit.

Insert the floppy disk you wish to copy in drive 1 and type:

```
dd if=/dev/rd1 of=/tmp/disk count=50 bs=440
```

This copies 2,300 blocks from the floppy disk in drive 1 to the file /tmp/disk. Then

remove the floppy disk from drive 1 and replace it with a formatted floppy disk. Type:

```
dd if=/tmp/disk of=/dev/rd1 count=50 bs=440
```

This copies the contents of the previous floppy disk to the new floppy disk. To make more than one copy, simply repeat the second dd command with additional formatted floppy disks.

When you are finished making copies, type

```
rm /tmp/disk
```

to erase the temporary file.

### Backing up your hard disk

Because your hard disk may malfunction accidentally and its data destroyed, you should periodically copy the information on it to floppy disks.

You can copy information to floppy disks by mounting each floppy disk (as described in *Using floppy disks* above) and using the cp or cpdir command to copy specific files or directories. However, the dump command provides a more convenient method of backing up entire file systems.

dump works with C900 COHERENT to dump to floppy disks rather than to magnetic tape. The S option specifies the size of a floppy disk in blocks. To dump the full (level 0) root file system on /dev/hd1 onto a nine sector-per-track floppy disk in drive 1, type the command

```
dump 01s /dev/rd1 2300 /dev/hd1
```

For a complete discussion of dump, see the *COHERENT Command Manual* and the *COHERENT Administrator's Guide*. You can also dump with the tar command.

### Normal shutdown

To preserve the integrity of your file system if you reboot the COHERENT system or turn off the C900 for any reason (other than repairing the root file system after performing the check -s command), you should always type the command

```
sync
```

before you reboot or power down the system. If you are running multi-user mode, sync is automatically executed every 30 seconds.

To reboot or power down a multi-user system, you should log in as root. Tell all other users to log off. Then use the ps -lex command to determine the status and owner of each process on the system. When user activity is ended and everyone else has logged off, type

```
kill -1 1
```

Note that the COHERENT system knows when to shift from standard time to daylight savings; no further adjustments on your part are needed.

#### Going to multi-user mode

After you have set the date and have received the '#' prompt, type <ctrl-D> (hold down the <ctrl> key and strike the D key simultaneously) to shift the system in multi-user mode. You should do this even if you are the only person to use the system. Multi-user mode has a number of advantages:

1. The system runs a process that updates the file systems every 30 seconds. This will largely prevent errors caused by power failure or other sudden system halt.
2. You may log in as a user other than root, and prevent serious damage to protected files in the event of a typographical or other error in a command line.

If you want to become the superuser, you can use the su command to become root. Just type

```
su root
```

and the password (if there is one) to become the superuser. If you are not the only user of this system, you must decide whether you want every user to be able to become the superuser; passwords can be used to control access to superuser status. For more information, see the *COHERENT Administrator's Guide*.

After you have gone to multi-user mode, the system will first execute the shell command file /etc/rc. etc/rc is executed when the system shifts to multi-user mode. The system will now say:

```
Coherent login:
```

If you want to change this login message, edit the file /etc/loginmsg. For example, you could change the login message to read

```
Joe's grille and Computer...
```

Now, you reply with your user name -- which can be any name in the /etc/passwd file, including root:

When you are want to log off, type <ctrl-D>; and the system will reply by printing the login message for the next time you want to use the computer.

#### 4. Tips on using the system

This section gives some more information on how you can use the COHERENT system to its fullest advantage.

##### Enabling cron

cron is a routine allows you to run commands on a regular basis. To use cron, simply type:

```
/etc/crontab
```

cron will then scan the file /usr/lib/crontab once a minute and execute all the commands you place there. For more information on cron, see the *COHERENT System Manual*.

##### Adding new users

To add new users John and mary to your COHERENT system, use the newuser command:

```
/etc/newuser john mary
```

newuser will create home directories for John and mary in the directory /u.

If you have a partition for users' directories (/dev/hd1 in the example given above), before you use newuser for the first time, you should be sure that /u is mounted by typing the command

```
/etc/mount /dev/hd1 /u
```

This will put the user files into the partition hd1. As shipped, this system mounts /u on hd1 when you bring it into multi-user mode.

newuser also creates a .profile file for each new user. When John's .profile is created, all it will contain is the line:

```
export PATH=/u/john/bin:$PATH
```

The .profile can be edited by the user himself or by the superuser to perform a number of tasks; for example, adding the command

```
/usr/games/fortune
```

to .profile means that the user will be greeted with a witty saying from the fortunes library every time he logs in.



to return the system to single-user mode. Make sure you remove floppy disks from the floppy disk drives *before* you turn off the power, or at least open the door of the disk drive. This ensures that if there is a power surge when the C900 is turned off, it will not harm the data on your floppy disks. Then type sync.

Now reboot the system by hitting the reset key, or simply turn off the power to your C900, wait approximately one minute, and turn your C900 back on again.

#### Named pipes

A named pipe, called a fifo (first in, first out), is a special device file that you can use to communicate between two processes.

To create an named pipe, enter the command

```
/etc/mknod name p
```

where *name* is the name of the pipe file and the letter *p* tells mknod that you are making a named pipe.

For example, to create a named pipe called *connection*, type

```
/etc/mknod connection p
```

To use *connection* to communicate between *process1* and *process2*, type

```
process1 >connection; process2 <connection
```

This is equivalent to typing

```
process1 | process2
```

The advantage of using a named pipe is that, unlike a normal pipe, the named pipe can be shared by processes that do not have the same parent, or are owned by different users.

Just like the regular pipe, the named pipe will fill up a buffer with the output from *process1* and pass it along to *process2* when the buffer is full. If you wrote the output from *process1* into a regular file and then look *process2*'s input from that file, *process1* would have to complete its work before *process2* started; with a named pipe, both can work simultaneously.

#### Differences from other documentation

The COHERENT system is available on a wide variety of computers. In almost every case, the operation of a COHERENT command is identical on each machine; the system looks the same to the user, regardless of which processor actually executes the commands. However, because of hardware limitations, a few commands do not exist or work differently in the C900 version. This section notes commands that are described in the *COHERENT Command Manual* or the *COHERENT Administrator's Guide* but which are not included in the current C900 version of COHERENT.

1. The erase character, which erases the last character typed, is `<backspace>` or `<ctrl-H>` by default, rather than `'@'` as noted in the COHERENT system's manuals. Similarly, the kill character, which erases an entire line, defaults to `<ctrl-U>` rather than `'@'`.
2. The spelling checkers spell and typo require large dictionary and data files. To keep this release within a manageable size, are not provided.
3. The manual command man is included, but the manual text in directory `/usr/man` uses large amounts of disk space and so is not sent with the standard release. They will be provided upon request for a nominal charge.
4. The magnetic tape handling program `lp` does not apply to the C900 and is not included.
5. The `xsmall` command has been eliminated.
6. The `newusr` command described above replaces the `mknusr` command mentioned in the *COHERENT Administrator's Guide*.

#### Possible problems

COHERENT commands can fail when your COHERENT system runs out of some resource, such as memory or disk space. This section lists a few of the problems you might see.

One error message is

```
Out of memory
```

This indicates that your COHERENT system did not have enough memory available to execute the command. It may mean that you have a program with an infinite loop that creates infinite output.

COHERENT may also tell you

```
can't find xxx
```

when you run out of memory, where `xxx` is the name of the command you typed.

The message

```
Try again!
```

indicates that the system reached its limit on the number of simultaneously active processes and therefore could not execute the command you requested.

The message

```
No space on (rri, n)
```

indicates that the system ran out of space on the file system for the device with major

number *m* and minor number *n*. You can find the major and minor numbers associated with each device by typing

```
ls -l /dev
```

You should unmount the file system and check it with the `-s` option to insure its integrity after you run out of space. If it is the root device, reboot immediately after running check, and check again after rebooting.

Many programs increase their data area automatically as needed. If a program grows too large, you may get the error message

```
bad system call
```

This usually indicates that COHERENT could not allocate enough memory for an exec system call. Sometimes it indicates that a program detected an irrecoverable error and terminated execution with the abort library routine, which deliberately issues a bad system call.

#### Touring the file system

In the section C900 COHERENT 0.7.3 file system below, you will find a list of the files on the distribution floppy disks. This section briefly describes each directory, to introduce the structure of the COHERENT file system.

The directory `/` is the root directory. All files in every file system are in the root directory or in one of its subdirectories.

Directory `/bin` contains the commands described in the *COHERENT Command Manual*. Some C900-specific commands are described in the *C900 Command Manual* pages below.

File `/coherent` contains the COHERENT system that is running on your C900.

Directory `/dev` contains data on devices.

Directory `/drv` contains data for some of the special device drivers.

Directory `/etc` contains files and commands used in system maintenance. The *COHERENT Command Manual*, *COHERENT System Manual*, and *COHERENT Administrator's Guide* describe these files and commands.

Directories `/f0` and `/f1` are empty directories used to mount floppy disks, as described under the heading *Using Floppy disks*, above.

Directory `/lib` contains the phases of the C compiler and the standard C libraries `libc` and `libm`.

Directory `/mnt` is an empty directory used to mount additional file systems.

Directory `/tmp` is used by many COHERENT commands to hold temporary files. The command file `/etc/rc` normally deletes all files in `/tmp` when you bring up the system in multi-user mode.

Directories `/u` and `/v` are for user files. `/u` is the default directory used by `newusr` for installing user home directories.

Directory `/usr` contains subdirectories for use by the users and the system. `/usr/adm` contains administrative files. `/usr/bin` contains the kernel and trout commands. `/usr/games` contains games. `/usr/include` contains header files included when compiling C programs. `/usr/lib` contains libraries and a few other files. `/usr/man` contains manual pages; because of the size of the *COHERENT Command Manual* and *COHERENT System Manual* files, they are not currently distributed with the system. `/usr/pub` contains an ASCII table. `/usr/spool` contains subdirectories for commands which save information for later execution, such as `lpr` and `mmail`.

## 5. C900 Coherent 0.7.3 files

This section contains a listing of the files on each of the three system disks for COHERENT 0.7.3.

## System disk 1

```

/bin
/bin/psdir
/bin/cat
/bin/ed
/bin/echo
/bin/check
/bin/lschack
/bin/dchack
/bin/od
/bin/do
/bin/lfile
/bin/ksdir
/bin/ls
/bin/vc
/bin/vsh
/bin/sync
/bin/l
/bin/bed
/bin/chgrp
/bin/chmod
/bin/chown
/bin/cmp
/bin/diff
/bin/du
/bin/dl
/bin/ae
/bin/rm
/bin/rmdir
/bin/egrep
/bin/grep
/bin/mv
/bin/pa
/bin/pwd
/bin/kill
/bin/build
/bin/lind
/bin/od
/bin/tty
/bin/login
/bin/sp
/dev

```

```

/dev/mem
/dev/kmem
/dev/nll
/dev/tty
/dev/ldi
/dev/hdd
/dev/hd1
/dev/hd2
/dev/rhd0
/dev/rhd1
/dev/tty50
/dev/tty51
/dev/lp
/dev/rhd2
/dev/swap
/dev/rhd3
/dev/hdd
/dev/modem
/dev/rfd1
/dev/tty52
/dev/ram
/dev/rhd12
/dev/rhd13
/dev/rhda110
/dev/rhda111
/dev/rhd10
/dev/rhd11
/dev/rhd12
/dev/rhd13
/dev/rhd10
/dev/rhd11
/dev/tty53
/dev/tty54
/dev/tty55
/dev/tty80
/dev/console
/etc
/etc/cifl
/etc/getty
/etc/init
/etc/load
/etc/ksks
/etc/ksksd
/etc/uload
/etc/umount
/etc/mount
/etc/group
/etc/rc
/etc/profile
/etc/passwd
/etc/boottime
/etc/.profile
/etc/mkproto
/etc/swap

```



```

/bin/dt1113
/bin/enroll
/bin/decode
/bin/encode
/bin/nccheck
/bin/spec1
/bin/erof1
/bin/man
/bin/ksort
/bin/lex
/bin/nld
/10
/emi
/emp
/v
/vu

```

System disk 3

```

/etc
/etc/mstab
/etc/mnttab
/etc/action
/etc/ern
/etc/update
/etc/wall
/etc/utmp
/etc/odate
/etc/profile
/etc/helpfile
/etc/lopmg
/etc/motd
/etc/news
/etc/newusr
/etc/helindex
/etc/termcap
/lb
/lb/cc0
/lb/ccl
/lb/cc2
/lb/cc3
/lb/cpp
/lb/erstd.o
/lb/dtoa.o
/lb/lbc.a
/lb/lbn.a
/lb/scrstd.o
/lb/yyparse.c
/lb/lby.a
/lb/albc.a
/usr
/usr/bin

```

```

/usr/.uberalt
/usr/spool
/usr/spool/at
/usr/spool/lpd
/usr/spool/mail
/usr/spool/pkexec
/usr/lb
/usr/lib/strun
/usr/lib/dt1113
/usr/lib/makemacros
/usr/lib/makexecfns
/usr/lib/dt111h
/usr/lib/dt11b
/usr/lib/ur11a
/usr/lib/binm1e8
/usr/lib/crontab
/usr/lib/lpd
/usr/lib/libmp.a
/usr/lib/spec1
/usr/lib/termc.am
/usr/lib/termc.a
/usr/include
/usr/include/sys
/usr/include/sys/types.h
/usr/include/sys/stat.h
/usr/include/sys/ftime.h
/usr/include/sys/intr.h
/usr/include/sys/proc.h
/usr/include/sys/fdisk.h
/usr/include/sys/inode.h
/usr/include/sys/const.h
/usr/include/sys/times.h
/usr/include/sys/time.h
/usr/include/rtime.h
/usr/include/access.h
/usr/include/action.h
/usr/include/ar.h
/usr/include/asser.h
/usr/include/canon.h
/usr/include/char.h
/usr/include/const.h
/usr/include/crypt.h
/usr/include/dlr.h
/usr/include/errno.h
/usr/include/fblk.h
/usr/include/llsys.h
/usr/include/lper.h
/usr/include/grp.h
/usr/include/intr.h
/usr/include/inode.h
/usr/include/l.out.h

```

```

/usr/include/math.h
/usr/include/mktime.h
/usr/include/mnttab.h
/usr/include/moon.h
/usr/include/mpr.ec.h
/usr/include/ms19.h
/usr/include/mstab.h
/usr/include/mtype.h
/usr/include/pod.h
/usr/include/setjmp.h
/usr/include/signal.h
/usr/include/set.h
/usr/include/set.h
/usr/include/sdio.h
/usr/include/lines.h
/usr/include/ty.h
/usr/include/types.h
/usr/include/vproc.h
/usr/include/vtmp.h
/usr/pdb
/usr/pdb/asc11
/usr/gsmas
/usr/gsmas/1b
/usr/gsmas/1b/rndll.m4
/usr/gsmas/1b/fortunes
/usr/gsmas/rndll
/usr/gsmas/moo
/usr/gsmas/fortuna

```

6. C900 COHERENT Manual Pages

This section contains additional pages for the *COHERENT Command Manual* and the *COHERENT System Manual* which apply only to the C900 version of COHERENT and to this release of COHERENT, as well as manual pages that have changed since the last major revision of the COHERENT manuals. The introductions in the *COHERENT Command Manual* and the *COHERENT System Manual* describe the format of these pages.

**NAME**  
cron--execute commands periodically

**USAGE**  
/etc/crontab

**DESCRIPTION**  
cron is a daemon that executes commands at preset times. The commands and their scheduled execution times are kept in the file /usr/lib/crontab.

Once each minute cron searches through the crontab. For each command stored there, cron compares the current time with the scheduled execution time and executes the command if the times match. When it finishes the search, cron sleeps until the next minute. Because it never exits, cron should be executed only once (customarily by /etc/rc).

The crontab file consists of lines separated by newlines. Each line consists of fields separated by white space (tabs or blanks). The first five fields describe the scheduled execution time of the command. In order, they represent:

minute (0-59),  
hour (0-23),  
day of the month (1-31),  
month of the year (1-12), and  
day of the week (0-6, 0 meaning Sunday).

Each field may contain a single integer in the appropriate range, a pair of integers separated by a hyphen '-' (meaning all integers between the two, inclusive), an asterisk '\*' (meaning all legal values), or a list of the above forms. The remainder of the line gives the command to be executed at the given time.

For example, the crontab entry

```
29 * * 7 0 msg henry Succotash!
```

means that every hour on the half-hour during each Sunday in July, the user named henry will have daemon: Succotash! written on his terminal.

cron recognizes three special characters and escape sequences in the crontab. If a command contains the percent character '%', cron executes only the portion up to the first '%' as a command and passes the remainder to the command as its standard input. cron translates any percent characters '%' in the remainder to newlines. The special interpretation of '%' can be prevented by preceding it with a backslash, '\%'. Finally, cron removes the sequence \newline from the text before passing it to the shell \$; this can be used to make an entry in the crontab more readable.

cron is designed for commands that need to be executed regularly. Temporal commands that need to be executed only once should be handled with the at command.

## FILES

/usr/lib/crontab for stored commands

cron

cron

## SEE ALSO

init  
COHERENT Command Manual: at  
COHERENT Administrator's Guide

cron

cron

NAME  
cu--terminal emulation and remote copy utility

USAGE  
cu [-h] [-l line] [-s speed] [profile]

## DESCRIPTION

cu allows terminal emulation and low-level copying of files between COHERENT and COHERENT-like systems. cu acts as a terminal on the serial line identified to COHERENT as */dev/modem*. It can be used to execute commands on a remote system, or to transfer files to and from the remote system.

The following command line options are available:

-l line Use 'line' instead of */dev/modem*  
-s speed Set the baud rate to 'speed'  
(default: 1200)  
-h Select half-duplex operation

If the file *profile* is specified, cu will read the file for commands and text, instead of taking input from the standard input device. Input lines that begin with the slide character '-' are commands, and must be one of the following:

-f Select full duplex operation  
-h Select half duplex operation  
-sspeed Set baudrate to speed  
-w Wait for carrier

After reading and executing command lines in *profile* (those beginning with '-') cu pushes the rest of the file onto a stack of input files, from which it reads commands and text. As cu reads the input file (or standard input after all other input files have been read), it writes lines not beginning with '-' to the serial port. Lines beginning with '-' are commands to cu. The recognized interactive commands are:

-# comment  
-! invoke an interactive shell on local system  
-<file redirect input from file  
->file redirect to file, as well as to standard output  
->>file redirect to file, but not to standard output  
->file append to file  
-> end output redirection  
-Pn pause n seconds  
-q exit cu and hang up phone line  
-Q exit cu without dropping phone line  
-f enter full-duplex (no echo) mode  
-h enter half-duplex (echo) mode  
-s n set serial port speed to n  
-Bfile get file from remote system  
-Pfile put file to remote system  
-? print a list of special commands  
-\_ send the character '\_'

cu

Mar 13 1985

cu

COHERENT Version 0.7.3 Preliminary Release Notes

Page 28

## NOTES

Lines of text that begin with '-' will interpreted as commands. If this is not what was intended, the single '-' must be replaced by "--" so that the '-' will be taken literally.

## DIAGNOSTICS

cu exits with non-zero status if an error occurs.



NAME  
 kermit--remote system communication and file transfer

USAGE  
 kermit *clb* *baud* *etc* *line* ]  
 kermit *rlu* *rlu* *baud* *line* ]  
 kermit *slab* *rlu* *baud* *line* ] *file* ...

DESCRIPTION

kermit allows the user to communicate with a remote computer system and to transfer files between the local and remote systems. kermit can transfer ASCII or binary files of any length in either direction. The two computers must be able to contact each other, such as through a serial line or by modem over a telephone line, and both systems must have kermit available. The user must have login privileges on both systems and appropriate permissions in directories used for file transfer.

The kermit command line specifies a *mode*, followed without intervening spaces by optional *flags*, perhaps followed by additional arguments and *files*. The three possible *modes* are:

- c connect the two systems so they can communicate.
- r receive files from the other system.
- s send each *file* to the other system.

kermit normally uses a default communication line at a default baud rate; the defaults vary on different COHERENT systems. It normally strips leading directory information from the pathname of each *file* it sends and converts the name to upper case; it converts the file name to lower case when receiving. The following *flags* modify its normal behavior.

- a Specify complete pathnames for sending and receiving files; used only with *s* mode. The *a* flag requires file names in pairs: first the file to be sent, then the receiving file. For example, the command  
 kermit sa /usr/joe/stuff.c /usr/tom/src/thing.c  
 sends the file /usr/joe/stuff.c but specifies its name as /usr/tom/src/thing.c for the receiving system. The target directory must exist on the receiving system. The *a* flag implies the *f* and *x* flags described below.
- b *baud*  
 Set the baud rate to *baud*.
- d Debug mode. Tells kermit to print messages (on the standard output, not the standard error) that describe its actions.
- e *etc* Change the escape character from the default '^' to the given *etc*; used only with *c* mode. The escape character marks commands to kermit *c* while it is running, as described below.
- f Suppress file name case conversion.
- h Host mode. Tells kermit to use the same line for file transfer and for communication; used with either *r* or *s* mode on the remote system only. When used with the *h* flag, kermit resets the line modes properly when it completes a file transfer. If the *h* flag is not used, it will probably leave

kermit

kermit

the remote system line in raw no-echo mode.

- l Image mode. Tells kermit to send a full eight-bit byte for each character; this is necessary to transfer binary (non-ASCII) files. If the *l* flag is used when sending, it should also be used on the receiving system.

- l *line* Use *line* to connect the two systems. For example, the command kermit clb /dev/tty50 1200  
 tells kermit to use line tty50 at 1200 baud instead of the default line and baud rate.

- m Macintosh mode. Necessary when sending files to an Apple Macintosh; used only with *s* mode.

- t Tymnet mode. Allows Tymnet to keep up with file transmission.

- x Allows the specification of a complete pathname for the receiving file; used only with *s* mode. For example, the command  
 kermit sx mydir/stuff

sends the file mydir/stuff to mydir/stuff on the receiving system. The target directory must exist on the receiving system and the user must have write permission in it.

kermit c recognizes two escape sequences. The default escape character '^' can be changed with the *e* flag, as noted above.

- ^c Exit from kermit and break the connection between the two systems. Note that this notation does *not* mean <ctrl-C>; rather, you must literally type a caret '^' and then the letter 'c'.

- ^s Suspend kermit on the host system but do not hang up the line.

Unlike some file transfer protocols, kermit requires that the user invoke kermit on both the sending and receiving systems to transfer a file. As shown in the example below, the user normally uses kermit c to connect to the remote system, invokes kermit with the *h* flag in either send or receive mode on the remote system only, types '^s' to suspend the local kermit c, and finally invokes kermit in receive or send mode on the local system.

The following example demonstrates the use of kermit. The example assumes the user is already logged in on the local system. The communication line is /dev/a10 and runs at 300 baud. The user wants to transfer locale to the remote system and remfile from the remote system. System names are in *italics* on the left, user input is in Roman, system responses are in bold, and remarks are in parentheses.

```

local          kermit clb /dev/a10 300          (connect to remote system)
local          kermit: connected...          (type a carriage return)
remote        Coherent login:                (perform login procedure)
remote        kermit shi remfile             (send from remote)
remote        )S-__@XaT                      (part of protocol, ignore)
remote        ^s                             (suspend local kermit)
local          kermit: suspended.
    
```

kermit

kermit

|        |   |                             |
|--------|---|-----------------------------|
| local  | kernel: rlib /dev/a10 300                           | (receive on local)          |
| local  | kernel: Receiving REMFILE as remfile                |                             |
| local  | kernel: done.                                       |                             |
| remote | kernel: clb /dev/a10 300                            | (connect again)             |
| remote | kernel: rbi   | (receive on remote)         |
| local  | kernel: suspend.                                    | (suspend local kernel)      |
| local  | kernel: silb /dev/a10 300 locfile (send from local) |                             |
| local  | kernel: Sending locfile as LOCFILE                  |                             |
| local  | kernel: done.                                       |                             |
| remote | kernel: clb /dev/a10 300                            | (connect again)             |
| remote | <ctrl-D>  | (log off the remote system) |
| remote | Coherent login:                                     |                             |
| remote | ^c  | (disconnect local kernel)   |
| local  | kernel: disconnected.                               |                             |

SEE ALSO

"Kernel: A File-Transfer Protocol for Universities," BYTE, June 1984 pp. 255 ff., July 1984 pp. 143 ff.

DIAGNOSTICS

Kernel may print the following error messages:

Aborting with following error from remote host:  
 Problem on receiving system.

Bad line speed: illegal baud rate.

Cannot create name: receiving system cannot create name.

Cannot open file name: sending system cannot open name.

Cannot open line: wrong line number.

No line specified for connection: line argument missing.

Receive failed: file not received.

Send failed: file not sent.

Speed setting not implemented: incorrect baud rate.

Yes, I'm still here...: connect command repeated.

NOTES

If you type kernel c and get the message kernel connected but the remote system does not respond, check the line that connects the two systems and the ability of the remote system to accept a login on the line.

Remember always to log off the remote system when done; kernel will not do it for you. If you do not log off, the next person dialing into the remote system over the same line will be logged in as you, without having to go through any password procedure.

The file transfer protocol uses small (96 character) checksummed packets, with ACK/NAK responses from the receiving system. The timeout period is five seconds, and kernel does ten retries before it abandons an attempted file transfer.

The kernel protocol was developed at the Columbia University Center for Computing Activities.

Tymnet is a trademark of Tymshare, Inc. Macintosh is a trademark of Apple Computer Corp.

kernel

kernel

Index

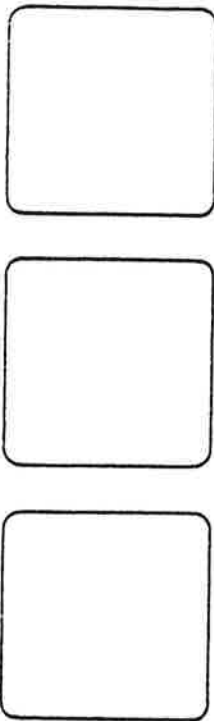
.profile: 10  
 /: 15  
 <ctrl-H>: 14  
 <ctrl-U>: 14  
 mkfs: 6  
 abort: 15  
 bad system call: 15  
 bla: 15  
 boot: 7  
 boot: 4  
 booting: 3  
 build: 4  
 building COHERENT: 4  
 can't find: 14  
 check: 7, 15  
 check file system: 7  
 coherent: 15  
 cp: 12  
 cpdir: 12  
 cron: 25  
 cron: 10  
 cu: 27  
 date: 8  
 dt: 11  
 dev: 15  
 device:  
 major: 14  
 minor: 14  
 drv: 15  
 dump: 12  
 erase character: 14  
 etc: 15  
 etc/boottime: 11  
 etc/logmsg: 9  
 etc/mlab: 11  
 etc/profile: 8  
 etc/rc: 9-10, 16  
 fd: 15  
 ft: 15  
 fifio: 13  
 files on floppy disks: 11  
 floppy:  
 disk drive: 3  
 disks: 11  
 format: 11  
 hardware requirements: 2  
 initial installation: 3  
 kernel: 29  
 kill: 12  
 kill character: 14

lib: 15  
login: 9  
login message: 9  
ls: 15  
man: 14  
memory: 4  
mkfs: 5, 11  
mknod: 13  
mkproto: 5  
man: 15  
mount: 10-11  
multi-user: 9  
named pipes: 13  
newusr: 10, 16  
no space: 14  
out of memory: 14  
password: 9  
PATH: 10  
pipes:  
  named: 13  
  power down: 12  
  prompt: 4  
  read error: 4  
  reboot: 12  
  root: 9  
  directory: 15  
set date: 8  
shutdown: 12  
software requirements: 2  
spell: 14  
sw: 9  
superuser: 9  
sync: 7, 9, 12-13  
tar: 12  
time zone: 8  
tmp: 16  
tp: 14  
try again: 14  
typo: 14  
u: 10, 16  
umount: 11  
usr: 16  
usr/adm: 16  
usr/bln: 16  
usr/games: 16  
usr/include: 16  
usr/lib: 16  
usr/lib/cronsh: 25.  
usr/man: 14, 16  
usr/pub: 16  
usr/spool: 16  
v: 16

**Table of Contents**

**Introduction to the**

**COHERENT System**



|   |    |
|---|----|
| 1. Introduction . . . . .                   | 1  |
| What is COHERENT? . . . . .                 | 1  |
| What is an operating system? . . . . .      | 1  |
| COHERENT design philosophy . . . . .        | 2  |
| COHERENT properties . . . . .               | 5  |
| 2. How do I begin? . . . . .                | 7  |
| Terminals and COHERENT . . . . .            | 7  |
| Special terminal keys . . . . .             | 7  |
| login — Logging in . . . . .                | 8  |
| Try COHERENT commands . . . . .             | 10 |
| Commands to COHERENT . . . . .              | 11 |
| help, man — Help with commands . . . . .    | 13 |
| Logging out . . . . .                       | 13 |
| 3. Features of COHERENT . . . . .           | 15 |
| Information storage and retrieval . . . . . | 15 |
| >, < — I/O redirection . . . . .            | 16 |
| — Pipes . . . . .                           | 17 |
| Processing information in files . . . . .   | 18 |
| Document preparation . . . . .              | 19 |
| Program writing tools . . . . .             | 20 |
| Electronic communication . . . . .          | 21 |
| Other COHERENT features . . . . .           | 21 |
| Summary . . . . .                           | 21 |
| 4. Files and directories . . . . .          | 23 |
| File names . . . . .                        | 23 |
| Your directory . . . . .                    | 23 |

|   |    |   |    |
|---|----|---|----|
| Pathnames . . . . .                                   | 24 | 6. Miscellaneous tools and features . . . . . | 53 |
| mkdir, cd, pwd -- More directories . . . . .          | 25 | units -- Convert units of measure . . . . .   | 53 |
| mv, cp -- Moving files between directories . . . . .  | 28 | bc -- Desk calculator . . . . .               | 54 |
| chmod -- File protection and mode . . . . .           | 31 | cal -- Calendar maker . . . . .               | 59 |
| rm, rmdir -- Removing files and directories . . . . . | 32 | crypt -- Secure information . . . . .         | 60 |
| du, df -- How much space . . . . .                    | 33 | 7. More about COHERENT commands . . . . .     | 61 |
| ln -- Linking files together . . . . .                | 33 | Simple commands . . . . .                     | 61 |
| Introduction to COHERENT commands . . . . .           | 35 | Special characters . . . . .                  | 61 |
| cat -- List contents of a file . . . . .              | 35 | Background commands . . . . .                 | 62 |
| >, <, >> -- Redirection . . . . .                     | 36 | C commands in a file . . . . .                | 63 |
| Lower case sensitivity in commands . . . . .          | 37 | .profile -- Login shell file . . . . .        | 65 |
| -- Pipes . . . . .                                    | 38 | Substitutions . . . . .                       | 65 |
| set -- List files on screen . . . . .                 | 38 | File name substitution . . . . .              | 66 |
| who -- Who is on the system . . . . .                 | 39 | Parameter substitution . . . . .              | 69 |
| ls, lc -- Listing your directory . . . . .            | 39 | Shell variable substitution . . . . .         | 71 |
| msg -- Send message . . . . .                         | 42 | Command substitution . . . . .                | 75 |
| msg -- Hear no messages . . . . .                     | 42 | Special shell variables . . . . .             | 76 |
| write -- Electronic discussion . . . . .              | 43 | * -- Read commands . . . . .                  | 77 |
| mail -- Mailing a letter . . . . .                    | 44 | test -- Condition testing . . . . .           | 78 |
| pr, lpr -- Print files . . . . .                      | 45 | Conditional command processing . . . . .      | 79 |
| echo -- Echo the command line . . . . .               | 46 | Control flow . . . . .                        | 80 |
| ed -- Text file editor . . . . .                      | 47 | Summary . . . . .                             | 86 |
| grep -- Find patterns in text files . . . . .         | 47 | 8. Creating and using programs . . . . .      | 87 |
| date -- Print the date . . . . .                      | 48 | Basic steps in COHERENT programming . . . . . | 88 |
| time -- Measure command execution time . . . . .      | 49 | ed -- Creating the program source . . . . .   | 88 |
| passwd -- Passwords . . . . .                         | 49 | cc -- Compiling the program . . . . .         | 89 |
| silly -- Changing terminal behavior . . . . .         | 50 | m4 -- Macro processing . . . . .              | 90 |

|  |     |
|--|-----|
| Programming simple input and output . . . . .      | 90  |
| make — Building larger programs . . . . .          | 93  |
| db — Debugging the program . . . . .               | 95  |
| Summary . . . . .                                  | 99  |
| 9. A sample problem solved with COHERENT . . . . . | 101 |
| Build a dictionary . . . . .                       | 101 |
| Maintaining the dictionary . . . . .               | 105 |
| Using the dictionary . . . . .                     | 106 |
| Summary of dictionary problem . . . . .            | 107 |
| Index . . . . .                                    | 109 |

## 1. Introduction

This document is an introduction to the COHERENT system. It has two purposes: one, to be a tutorial manual; and two, to be a reference manual for the COHERENT system.

A related manual is the *COHERENT Command Manual*. It contains detailed descriptions of each command in the system. The *ed Interactive Editor Tutorial* is useful to those who have not used the COHERENT text editor ed. It also contains general information about the system. The *COHERENT Administrator's Guide* is useful to the person or persons responsible for bringing up the COHERENT system and maintaining it.

Sections two through nine contain specific details on the use of the COHERENT system. The remainder of this introductory section describes what an operating system is and outlines the philosophy of the COHERENT system.

### What is COHERENT?

The COHERENT system is a timesharing operating system that can be used on many hardware configurations. Its operation employs a few elegant concepts giving a powerful and flexible system which is still simple to use. These simple concepts have the same importance to the practice of software development as interchangeable parts had to the industrial revolution.

### What is an operating system?

When you use a computer, you will use programs and data. A program is a series of instructions to the computer to direct it to perform a task, such as playing a game like tic-tac-toe, or balancing a checkbook. A personalized telephone directory is an example of data.

Unless the computer you use is very small, you will be sharing its resources with other people. Resource sharing not only yields many economies but also allows many people to communicate with each other and thereby work more efficiently together. At the same time, users are protected against unwanted interference from other users.

The elementary information-processing functions are carried out by the computer hardware—the boxes, circuits, and wires. However, software is necessary to provide the complex set of features that makes the computer do what you want it to do.

This software is called a *timesharing operating system*. Any timesharing operating system must be able to:

- schedule computer time
- use mass-storage devices
- organize disk storage space
- protect programs from unwanted conflict
- protect stored information from destruction
- facilitate cooperation with other users
- In sharing programs and data

In short, a timesharing operating system is a set of programs that enables many computer users to share the hardware and software resources of a computer efficiently.

Today's computers would not be truly general purpose without an operating system. Additional tools are often provided as part of operating systems. Editors, compilers, debuggers and assemblers are necessary for you to develop and test programs. Document preparation aids greatly facilitate your creation of memos, manuals or even books. Command processors assist you in controlling the computer and solving your problem. Status checking programs tell you the level of system usage, disk space usage, and which other users are on the system.

The combination of the basic operating system features and the additional tools transforms the collection of wires, silicon, circuit boards and oxide-plated surfaces into a useful computer.

#### COHERENT design philosophy

COHERENT has all the properties that have been discussed so far. But the quality and quantity of the features provided by the COHERENT operating system distinguishes it from other operating systems.

The quality is guaranteed by the strict use of state-of-the-art software technology. All but a very small part of the operating system software is written in C, a high level language, rather than

assembler. The result is a very reliable operating system, with no observable loss in execution speed.

The choice of a high-level language also provides portability. The C language already exists on a number of popular computers, and is certain to become more and more available. At this writing it is available on the Z8000, PDP-11, the 8086, and the 8088, with the 68000 and other popular machines in development.

An important guiding principle in the design and implementation of the COHERENT operating system is that good performance is the direct result of dedication to careful design and implementation of algorithms and systems, rather than refuge in coding tricks.

The basic features noted above embody the purpose of the operating system at the most primitive level. It is instructive to examine the necessity and utility of providing these features.

A computer system is not an end in itself but an instrument used to implement solutions to real-world problems. It is required to be a tool bench that provides for the construction of other tools with a myriad of specific applications.

If the design of the operating system is too specialized or limited, it will only suit a few applications. On the other hand, if too much specific detail is put into the operating system itself, then the system becomes very complex, difficult to use and maintain, and potentially quite unreliable.

The design philosophy of COHERENT is expressed well in this quote from John Conway\*: "The engineer who wants a machine for some specific purpose will normally approve the simplest machine that does the job. He will not usually prefer a multiplicity of parts with the same effect, nor will he countenance the insertion of components with no function." Machines whose designs are based upon fundamentals are far more likely to satisfy these criteria.

The COHERENT system follows this approach throughout. For example, consider device-independent I/O. No distinction is made between a program, a device (such as a terminal or floppy disk),

\*J.H. Conway, *Regular Algebra and Finite Machines*; Chapman and Hall Ltd., London 1971, page 3.

and a file. Programs can transfer data between devices and files without knowing any of the physical characteristics of the device. This device independence results from basing the design of the I/O system on a consistent view of files, devices, and programs. All look to each other like a stream of bytes, so they can all communicate with each other directly.

If an application requires a more complex file structure such as an indexed sequential file or a B+ tree, such a structure can be added at a higher level. This greatly simplifies the design of the operating system, sparing unnecessary overhead for programs operating at the byte-stream level.

You might wonder at this point about a possible loss of efficiency or performance compromise within this design. To the contrary, the speed at which the COHERENT system transfers data between files on a disk is very nearly the hardware speed of disk-to-disk transfers. This is achieved through the use of simple but ingenious algorithms.

With the consistency of design exemplified by the device-independent I/O applied throughout the COHERENT system, a few primitive operations easily provide communication between programs, files and devices. With these, any user of the COHERENT system can construct building blocks which can be readily assembled to build a solution to a problem.

In the evolution of the classical operating system, features were added to the basic operating system and its programs in an *ad hoc* fashion. To solve a previously unanticipated problem, some existing program was modified to add a new feature. The new feature was then selected by a switch set by the user.

This approach created an operating system whose programs grew larger and more complex. As a result, the system was more difficult to understand and maintain and on the whole, less reliable. For example, a typical file-to-file copy program provided with a widely-used classical operating system may have three dozen options. In the COHERENT operating system, there is no comparable program. To copy files, the program `cat` is used with its output stream redirected. `cat` is the COHERENT command that copies a file to the user's terminal.

*cat* is a *filter*. A *filter* is a program that produces on its output a possibly modified copy of its input. A user can use existing filters, alone or in combination, or can easily construct new ones, to achieve the variety of actions that are elicited by *switches* in other operating systems.

This modular approach gives the COHERENT system user a great productivity advantage, as well as making the system more reliable. A program or filter designed for use in one application can be used in many other applications, even those which were not anticipated by the developer. Commands, even powerful ones, tend to be simple.

For example, you might want to know how many people are using the computer. The command `who` will produce a list of user names, their terminal designation, and the time of day and date that they logged in. Each user is listed on a separate line.

This may not be the information you want. If you only want to know how many users there are, the filter named `wc` (for word count) will help you find out. In the form

```
wc -l
```

we will count the number of lines in the input. Combining these two programs with the vertical bar *pipe* operator, the command

```
who | wc -l
```

will tell how many users there are on the system. The pipe connects the output of the `who` to the input of the filter `wc`, whose output will appear on the terminal.

The pipe feature has many applications throughout the use of the COHERENT system and increases its power substantially.

### COHERENT properties

The COHERENT file system has a tree-structured directory. This means that directories may contain files, which in turn may be data files or directories. The fact that directories may contain more directories can be a significant help in managing large numbers of files.

The COHERENT operating system is modularly designed using certain mathematical concepts. This results in a much better design



for the system. Using this simple but elegant approach, features are designed to fit well together. This means that COHERENT does not repeatedly reinvent the wheel—each feature is carefully designed to function well by itself and work readily in combination with others. COHERENT avoids the “creeping feature” syndrome common to classical operating systems.

An example of this modular design principle can be found in *character patterns*, or simply *patterns*. Many parts of the COHERENT system use patterns to describe strings of characters in a general way. Rather than having each part of the system specify strings in a different fashion, standard pattern specifications are used.

Patterns simplify specification of arbitrary strings of characters. For example, you can specify all strings of characters beginning with the letter *n*, containing one other vowel, and ending with the letter *r* with the pattern

```
n.*[aeiou].*r
```

## 2. How do I begin?

This section covers how to get on the system the first time, and is of most interest for those who have not used the COHERENT system before. It is also useful to those who may need to review the basics.

### Terminals and COHERENT

You will use a *terminal* to send commands to the COHERENT system and view its responses.

There are two kinds of terminals in use today. Each type of terminal has a keyboard. The keyboard is like that of a typewriter with a few special keys.

The older kind of terminal resembles a typewriter. This kind of terminal is called a *hard copy* terminal. The information you type or that the COHERENT system sends back is printed on paper.

The second kind of terminal widely available today, uses a screen resembling a television screen, called a *video display* or *CRT*. The purpose of the display is not to show pictures, but to display the dialogue between you and the computer system.

On a typical video display terminal there are 24 lines of characters visible on the screen. Each line on the screen can contain up to 80 characters.

All the work you do with the COHERENT system will be done by typing commands and data on the terminal.

### Special terminal keys

One special key on the keyboard that you will use in your work with the COHERENT system is the <RETURN> key. This key signals COHERENT that the end of a line has been reached, and that you want it to process a command. Not all terminals label the key <RETURN>, but each terminal has a similar key. Some will call it *newline*, *linefeed*, *enter*, or *eof*, and the key is usually larger than other keys (except the space bar). From here on, this key will be called <RETURN>.

Note that commands to the COHERENT system will end with a <RETURN>. No action upon your command will take place until you end a line with this key.

Another special key is the control key, usually labeled `ctrl` or `cntl` or `cont`. Most terminals place it on the left side of the keyboard. This is an important key used in sending certain special characters.

To use the `ctrl` key, you must hold it down while you press another key. To send the computer a `ctrl-D` character, hold down the `ctrl` key, strike the `D` key simultaneously, then release both keys.

Since there is usually no printable character corresponding to control characters, in this document they will be represented in the form:

`<ctrl-D>`

for the character `ctrl-D`.

While you are typing information to the COHERENT system, you can correct the information before it is processed. There are two keys that help you do this. The first is the `<KILL>` character, which will erase the line entirely, and allow you to begin again. This key is usually the `@` key, but you can easily change the `<KILL>` character with the `stty` command, which is discussed in a later section.

The other key is the `<ERASE>` character, normally the `<ctrl-H>`. This will erase the most recently typed character. You can erase several characters with `<ctrl-H>` by striking it several times.

One more special key is the `<INTERRUPT>` key. This key can be used to halt a command in progress before it normally terminates. This key may be labeled `rubout`, `del`, or `delete` on your terminal.

#### login — Logging in

Before you use the COHERENT system for the first time, you need some specific information about your installation and your access to the COHERENT system. This information will come from your project leader, or the system administrator. If you have any questions about logging in or other COHERENT system topics, ask this person.

First, the administrator will tell you what your *user name* will be. User names are typically first names, initials, nicknames, or last names. The COHERENT system recognizes you by your user name

while you are using the system. Others who are using the system will use this name to communicate with you.

Secondly, the administrator will tell you what your password, if any, will be. This password is important for the security of the entire system, and you should not divulge your password to others. If your installation does use passwords, you will need to know yours before you get on the system.

Once you have this information, the first thing you must do is to *log in* to the COHERENT computer system. Doing so will establish the connection with the computer and ready the system for your commands.

In order to log in, you must first determine if your terminal is *hard wired* to the computer, or whether you must make a phone call to establish your connection. If you do not know, ask the system administrator.

If your terminal is not hard wired to the system, your system administrator will supply you with a telephone number of the computer and instructions on how to connect your terminal to the telephone.

If your terminal is hard wired to the COHERENT system, your first step after turning on the terminal is to send the `<ctrl-D>` command.

Once you are connected to the COHERENT system, it will reply either:

login:

or

Coherent login:

In response to this, type your user name, followed by a `<RETURN>`. If you have a password, COHERENT will prompt you to enter it by typing:

Password:

When you type your password, the numbers and letters will not be shown on the screen or typed on the paper. This is to prevent unauthorized users from seeing your password. Follow the pass-

word with a <RETURN>. If you entered the password incorrectly, COHERENT will ask you to try again.

If you have entered your password correctly, you are now logged in. You will be greeted by the message of the day, if there is any.

Once your login is successful, the COHERENT system is ready for your commands. To indicate readiness, COHERENT sends a *prompt* character to tell you. In most installations, this is a dollar sign:

```
$
```

You are now ready to type commands. When a command is finished, COHERENT will issue another prompt, meaning that the system is ready for your next command.

#### Try COHERENT commands

To see how easy it is to use COHERENT, type the following lines. Be sure to end each line with a <RETURN>.

```
ed
l
This is a sample COHERENT file.
w file01
q
```

The characters `ed` tell COHERENT to call in the editor program, which is used to build and change files. The information that you type will then be processed by the COHERENT editor. When you are finished with the editor, you return to COHERENT by typing the `Q` or `q` command. Now type

```
cat file01
```

This command types out the contents of the file `file01` that you just created. Finally, type

```
lc
```

This command lists the files that you have, and will reply

```
Files:
file01
```

Congratulations! You have just made COHERENT work for you.

The first command `ed` created a file and filled it with some text, while the second command `cat` typed the file out on your terminal. Finally, the `lc` command listed the name of each of your files. See following sections for full descriptions of each of these commands.

#### Commands to COHERENT

Once you have logged into the system, all the resources of the COHERENT system are at your fingertips. COHERENT commands give you control over these resources.

COHERENT is easy to learn and well thought out. The best way to learn the system is to try examples shown here and elaborate upon them. The more things you try, the more you will learn and the more proficient you will be in using the COHERENT system. If there are other users that you can talk to about the system, you may find it helpful to exchange information with them.

All commands have common elements. Commands consist of two parts. The first part is the name of the command itself.

Some commands only have the first part. For example, to list the names of files that you have, type

```
lc
```

and COHERENT will print their names in columns across the screen.

A file is a set of information stored on disk. Files are described in detail in Section 4.

If you have no files, `lc` will not type anything. If you have logged in for the first time, you may or may not have files, depending upon your installation. Try it. In any event, COHERENT will prompt you for another command after it finishes `lc`.

The second part of a command is a list of *parameters* or *arguments* to that command. We may think of parameters as controlling the behavior of the command, or as the target of the command's action. Each command must be terminated by hitting <RETURN>. Spaces or tabs separate the parts of the command.

The parameters of the command can be further divided into *options* (or *controls*) and *names*. Names are most frequently file names.

The options change the action of the command. Options are indicated in the command line by prefixing the option with a '-' character.

An example of a *name* parameter is shown in this example of a *cat* command:

```
cat file01
```

This command will type the information in *file01* on your terminal. The *name* argument is *file01*.

The command lists your file names one name per line.

```
ls
```

will produce a list in the form:

```
.profile
compu
file01
mailbox
```

However, there is more information available about each file in addition to its name. In response to the command

```
ls -l
```

ls will print this additional information. The '-' signifies an option. And the option letter l means "produce the long listing". This listing shows the size of the file, the date of creation or modification of the file, and the degree of protection of the file:

```
-rw-r--r-- 1 you      17 Sat Aug 15 17:20 file01
```

Another example of a *name* parameter to a command is a file name.

If you want to modify an already existing file, use *ed* with a parameter giving the name of the file.

```
ed file01
```

This will allow you to change the file created with the *ed* command above. To exit from *ed*, use the *Q* command.

As each command is discussed, the parameters to be used with it will also be discussed. Notice that the name of the command is

separated from the parameters by a space, as the parameters are from each other.

**help, man — Help with commands**

The COHERENT system has a help command

```
help
```

which will give you a brief description of COHERENT commands. To introduce yourself to these commands, type *help* by itself, or

```
help help
```

Both will tell you how to use the help command. To get information on the *lc* command, type

```
help lc
```

To obtain detailed information on a command, use the *man* abbreviation for *manual* command. On most COHERENT systems, each command has descriptions online which the *man* command will print out for you. To find out about the *man* command, type

```
man man
```

If your CRT screen fills with information, *man* will wait for you to type <RETURN> to continue. This is to prevent you from missing information should it scroll too fast. *man* will also wait for a <RETURN> after it puts out the last line of the description.

The command descriptions provided by the *man* command are available in printed form in the *COHERENT Command Manual*. It provides a concise description of each available command.

**Logging out**

When you are finished using the computer, you need to tell the COHERENT system that you are done, and free the terminal for other use. This step is called *logging out*.

There are two ways to log out. The first is to type a <ctrl-D> when COHERENT is expecting a command. The second is to type the command

```
logln
```

which will log you out and prepare for another login. Either way, you can then turn off your terminal.

### 3. Features of COHERENT

This section presents some basic concepts, such as files, directories, and pipes, which are important in understanding and using the COHERENT system.

#### Information storage and retrieval

Computer systems store information in *files*. A file on the computer is similar to the files you find in an office filing cabinet. All operating systems provide programs to help you create and use files. There are many different ways that file systems are designed. Files reside on the hardware called a *disk*. A file, once created, may be referred to, changed, or removed. The COHERENT system keeps each individual file as a separate entity. Much of your work with the COHERENT system will be based upon files.

To keep track of files, you need something that performs the same function as the index tabs on a file folder. A *directory* is COHERENT's way of doing this. The directory holds the names of files and marks where the files are located so that the COHERENT system knows where to find them. You will use the directory to keep track of your files.

As a user of the COHERENT system, you are not limited to one directory. You may have as many as you wish, as long as you don't run out of disk resources.

Directories for COHERENT are tree-structured. Your directory is a file in a parent directory. The following example will clarify this concept.

If you have three separate projects, and each has files of its own, then you can set up your directory to look like this:

```

                                yourname
proj1                            proj2                            proj3
source.1                        source.3                        proj3n source.k
source.2                        object.4                            3n.source

```

proj1, proj2, and proj3 are all subdirectories in directory yourname. Another level of subdirectory is with proj3a in subdirectory proj3.

Each user of the COHERENT computer system has his own directory. The COHERENT system makes sure that you automatically use the directory created for you and not that of other users. Similarly, your files are protected from accidental damage by another user.

However, if you wish, you can allow other users to examine or change your files.

Whether or not others may examine or change your files depends upon the type of *protection* that you choose for your file. In the usual case, you will not explicitly specify any protection, and the COHERENT system will create the file unprotected. Since directories are also files, you may prevent other users from examining the file names in your directory or subdirectory using the same protection mechanism.

Files in the COHERENT system contain several different kinds of information, ranging from programs to electronic mail. Later sections will present examples of each kind of file.

#### >, < — I/O redirection

Typical COHERENT commands write their output to the *standard output* file, which is normally your terminal. For example, who prints the name of each user currently logged into the system on your terminal:

```
who
```

By using the special character >, you can place this output in a file. The command

```
who >whoFile
```

will put this information into whoFile. The operator > tells COHERENT to redirect the standard output. Later, you can list the information on your terminal using cat:

```
cat whoFile
```

Once the information is in a file, you can process it in other ways, like sorting:

```
sort whoFile
```

will type the users' names on your terminal alphabetically.

Similarly, the *standard input* may be redirected to accept input from a file rather than from your terminal. The command we will count words, lines and characters from the standard input. Using input redirection, signaled by <, you can do the same processing for a file, such as whoFile:

```
wc <whoFile
```

#### | — Pipes

An important feature of the COHERENT system in building modular solutions is the *pipe*.

Pipes and truly device-independent I/O provide COHERENT with a concept as important as interchangeable parts. Programs can communicate with each other easily, even though they are independently created.

Most COHERENT programs are written as *filters*. A filter is a program that processes its input sequentially and produces sequential output. Generally, no titles or end of job messages are produced by filters.

Designing programs as filters gives greater flexibility in connecting programs with each other, as in the example where the output of who is processed by wc.

The mechanism that connects filters together is called a *pipe* and is indicated by a vertical bar:

```
|
```

The pipe operator in the command

```
who | wc
```

takes the standard output from `who`, normally destined for the terminal, and connects it to the input of `wc`, which would otherwise get its input from the terminal.

This command performs the same operation as shown in the section on I/O redirection:

```
who >whofile  
wc <whofile
```

except that `whofile` is a file that is to be removed later. The pipe is much handier to use and does not require you to remember to clean up temporary files like `whofile`.

The power and flexibility of the COHIERENT operating system owes much to the pipe.

#### Processing information in files

This section outlines some tools that COHIERENT provides to process data files.

Computer applications, such as general ledger, mailing label processing, accounts receivable processing, and inventory control, center around data files and transactions involving them. If you are building or supporting a software system, it may be productive to put project control information on the computer.

COHIERENT has data file processing capabilities that can help you implement such an application easily. Many of these commands will be directly useful to you.

`sort` can be used to order the lines or *records* in a file. By specifying options, you can sort a file based on any *field* or set of fields in each line, as well as select the field separator. You can also discard elements that are not unique.

Several input files may be sorted into one output file, thereby giving a merge capability. The files to be merged need not be previously sorted.

`awk` is a pattern scanning and processing language for the COHIERENT system. It can be used to write reports, to detect patterns in files and to do online data entry validation. `awk` treats its input as lines consisting of fields. `awk` supports numeric as well as string processing on the same fields. Totals and averages can be easily computed on any of the input fields. Associative memory arrays are provided, where array indices may be integers, strings, or even floating point numbers.

If you have two text files that contain almost the same information, you can discover exactly what the differences are with the command `diff`. This can be useful in illustrating changes to a document between versions in showing how today's inventory file relates to yesterday's inventory file. The command `diff` can also help you to find differences between two versions of a contract under negotiation—your original and the one returned to you, perhaps changed, by the other party. Used in conjunction with `ed`, `diff` can help you maintain one master file and a series of automatic update commands to produce other versions of the file.

A similar program `cmp` can process non-text files.

A related program `comm` will process sorted files and show you the similarities they share.

The command `uniq` inputs a sorted file and outputs a file with duplicate lines removed.

You can use `grep` to find patterns in text files.

These commands can be combined to derive many kinds of information easily.

#### Document preparation

The COHIERENT system can be used for document preparation as well as program development. It has been used for word processing applications, computer program documentation, and many user manuals. With COHIERENT, you can write a manual that tells other people how to use a program. You can write manuscripts as large as a book, or as small as a memo.

By means of commands embedded in a text file, you can use the command `nrff` to format your document attractively. You can set margins, paragraphs, and page headings; `nrff` will right-justify the lines of output text by appropriate padding of blanks between words.

You first enter the basic document text with `ed`. Then the text is given to `nrff`. If changes are necessary, you only need to enter the changes using `ed`... you do not need to retype the entire document.

`nrff` is very flexible. It is built with a large number of basic commands as well as the ability to add more commands. In fact, when you write a simple memo or a manual, you are using a small set of extended commands provided as part of the COHERENT documentation package.

If your need is truly sophisticated, you can add your own commands to `nrff` to affect nearly every aspect of the final appearance of the document. `nrff` can help you do this in such a way that a manuscript may appear in any of several different formats, without changing the content of the manuscript. Also, `nrff` can produce output that is used directly in typesetting.

#### Program writing tools

Writing programs is easy on the COHERENT system. The fundamental design of the system produces tools suited not only to accomplishing your desired task but also to provide a superior environment for program development.

The COHERENT system has a host of high-level language compilers. To assist in the debugging of programs, symbolic debuggers are provided for many of these languages.

The languages currently provided with COHERENT are:

- C
- assembly language

Pascal will be provided in the near future.

#### Electronic communication

COHERENT has several features that can provide electronic communication.

You may communicate with other users currently logged into the system with the `msg` command. `write` is similar to `msg` but allows extended dialogue.

To communicate with someone not currently logged into COHERENT, you can use the `mail` command.

#### Other COHERENT features

COHERENT provides many interesting tools. The program `units` converts different units of measure. To perform a calculation using your terminal, you can use the desk calculator program `bc`. To see the calendar for year, you can use `cal` to print a calendar on your eyes with the program `crypt`. These tools and others are discussed in detail in Section 6.

#### Summary

The COHERENT timesharing system provides a powerful base for the manipulation of information. Its file system has a structured layout. Input and output redirection and pipes enable the construction of program modules that can be easily recombined for additional flexibility and power.

The COHERENT system contains many commands that manipulate information in files, as well as tools that assist you in writing programs.



#### 4. Files and directories

In earlier sections, we introduced files as the cornerstone of the COHERENT information storage and retrieval capability. This section will discuss the topics of files and directories in more detail.

##### File names

Each file has a name, such as:

```
.profile  
file01  
cmd.sh  
file01  
test.c
```

File names are generally made up of upper case and lower case letters and numbers. COHERENT treats capital letters differently from lower case letters. The two file names `file01` and `file01` are therefore different.

A recommended set of symbols for file names is the lower case alphabet, the upper case alphabet, decimal digits and punctuation marks:

#

The file name should not be more than fourteen (14) characters long. If you specify a longer name, characters beyond the fourth will be ignored without any error message.

##### Your directory

The COHERENT system keeps your directory of file names current. You can inspect the directory with the `ls` and `lc` commands. When you specify a file name, COHERENT looks it up in the directory and connects the file to the program using it.

There are many directories on the COHERENT system. When you log in to the system, COHERENT sets up your home directory, which is determined by the system administrator.

You may sometimes need a program or a data file in another user's directory. Also, the commands that you use frequently come from another directory.

To examine or use files in a directory other than your own, you will need to specify the name of the directory as well as the name of the file. Separate the parts of the name of the directory by a slash:

```
/
ls /usr/henry
```

if the other user's name is *henry*.

### Pathnames

The tree-structured nature of the COHERENT file system means that all files in the system branch from a common origin. The origin is called the root. The name of the root directory is

```
/
ls /usr
```

One file in the root directory is *usr*. This is a subdirectory that normally contains the directories of all users. To list the names of all user directories, type the command:

```
ls /usr
ls /usr/henry
```

If one of the user names is *henry* as above, the command

will list the names of the files in *henry's* directory.

The parameter */usr/henry* is called a *pathname*. Pathnames may be fully or partially specified. All fully-specified pathnames begin with */* for root, and continue with further subdirectory names.

Pathnames that do not begin with a slash are *partially* specified, and are automatically prefixed with the current directory pathname to make them complete before use by the system.

Parts of pathnames are separated by slashes, so if there were a file in newfile directory named *newfile*, you would refer to it as

```
newfile/newfile
```

The absence of a beginning slash indicates that the pathname begins in the current directory. Thus, if your home directory name is

*henry*, then an alternate but less convenient way to specify the pathname to *newfile* is

```
/usr/henry/newfile/newfile
```

Thus, a pathname is a list of all the subdirectories leading from the root to the file in question. *newfile* is a file in subdirectory *newfile*, which in turn is a file in the home directory *henry*, which is further a file in the directory *usr*. The directory *usr* is a file in the master or root directory for the system.

You don't need to specify all of this, fortunately, whenever you want to specify a file in a subdirectory.

Partially specified pathnames are presumed to be within the current directory. Therefore, you can specify a subdirectory by specifying the name of the directory first, followed by the rest of the pathname.

### mkdir, cd, pwd — More directories

You can easily create more directories within your primary, or home directory. You may in fact create several directories, or even more subdirectories within them.

Some COHERENT users will create subdirectories for program source, documentation, completed programs, and commands. This can help locate a single file among many. Additionally, old versions of documents or programs can be kept in a separate directory.

Create *file01* using *cat* by typing:

```
cat >file01
This is another sample file.
<ctrl-D>
```

Now, you can use the copy command *cp*:

```
cp file01 file02
```

creating *file02*. Then *ls* will show

```
files:
file01 file02
```

You may have other files present when you log on the first time, depending upon your installation.

To create a new directory named `newdirectory`, use the command `mkdir` in this fashion:

```
mkdir newdirectory
```

If you follow this command with `ls`, it will list your regular files, but it will also list `newdirectory` separately as a directory:

```
Directories:
  newdirectory
Files:
  file01      file02
```

To refer to files, use this new directory name in specifying the path-name.

Now, create a file in the new directory by typing

```
cat >newdirectory/newfile
lines to be
contained in newfile
<ctrl-D>
```

This command copies lines to the file described by the partial path-name `newdirectory/newfile`.

A way to avoid specifying all of the subdirectories in a long path-name is to change the *current* (or *working*) directory. When you first log in, the current directory is set to your *home* directory.

If you have used the command `cd` to change your current directory, you can remind yourself what the current directory is by using

```
pwd
```

which means "print working (or current) directory". If you have a subdirectory backup in your directory, and change directories with

```
cd backup
```

then a command

```
pwd
```

will respond with

```
/usr/yourname/backup
```

The command `cd` (for change directory) will change the current directory. To change to `newdirectory`, issue the command

```
cd newdirectory
```

Before this command, your current directory was `/usr/henry` if your user name is `henry`. If you type the command `pwd`, the new path-name will be shown to be `/usr/henry/newdirectory`.

Now, if you issue an `ls` command, the listing will include only

```
Files:
  newfile
```

since `ls` with no parameters lists the current directory.

To change back to the directory that you had when you logged in to the system, use the `cd` command with no parameters:

```
cd
```

This directory is often referred to as the home directory. To change to another user's directory, you would say

```
cd /usr/other
```

or use the abbreviation

```
cd ../other
```

Here `..` is a special COHERENT system abbreviation for parent directory, which in this case is the `/usr` directory. In other words, `..` stands for the directory in which the current directory resides. Every directory in the system except the root directory has a parent. For the root directory, `..` refers to itself.

Another directory abbreviation is `..`, meaning current directory.

Assuming that your user name is `henry`, and your current directory is your home directory, part of the file system structure is

bin           usr(..)       etc

henry(..)     other

Here '..' is /usr, the parent directory path, and .., the current directory pathname /usr/henry. Both '..' and '..' change when you issue the cd command.

To see what your current directory is, you can use the command

pwd

(For print working directory) and COHERENT will reply with the full description of your working directory name. For example, if your user name is henry, and your installation uses the user name as the directory name, then pwd will reply

/usr/henry

**mv, cp — Moving files between directories**

Once you have created your new directory, you can move files to it with mv, or create new files there with commands such as ed.

To move file01 to newdirectory, the mv command is useful.

mv has two parameters. The first is the file to be moved; the second is either the new name of the file, or the destination directory of the file. So, to move file01 to the new directory, you can say

mv file01 newdirectory/file01

In this case, both parameters are file names. Alternatively, the second parameter can be a directory pathname:

mv file02 newdirectory

The second parameter is the directory that is to contain the file, and the name of the file in newdirectory will be the same as it was in the current directory. These two forms have the same effect.

To see where the files are now, type the two commands:

ls  
ls newdirectory

The result will be:

Directories:  
newdirectory

followed by

Files:  
file01 file02 newfile

To move the files back, use a combination of the commands already shown. Type

mv newdirectory/file01 file01  
cd newdirectory  
mv file02 ..  
cd

You can copy files with the cp command. This command has two parameters. The first parameter is the file to be copied, and the second parameter is the pathname of the new copy. To copy file01 to nfile01 in newdirectory, type the command

cp file01 newdirectory/nfile01

The difference between mv and cp is that after the cp command, both the original file and the copy exist, while after mv, only one copy exists.

Now, an illustration of what has been discussed so far about directories and files with an example.

Continuing with the user name of henry, assume that you have some documents that you have entered with ed, and you want to make backup copies of these files for safekeeping. The document file names are doc1 and doc2 and are in your home directory. For the purposes of this example, create doc1 with cat by typing:

```
cat >doc1
a few
lines of
text
<ctrl-D>
```

and similarly doc2:

```
cat >doc2
second file
with some text
<ctrl-D>
```

Don't forget that <ctrl-D> means to hold the control key down and simultaneously type the D key. An `ls` command tells you what your file names and directory names are:

```
Directories:
newdirectory
Files:
doc1 doc2 f11e01 f11e02
```

The first step is to create the directory to hold the backup copies. To help remind yourself what the directory is for, name it `backup`.

```
mkdir backup
```

Then, `ls` will show you:

```
Directories:
backup
Files:
doc1 doc2 f11e01 f11e02
```

Now, you can use the `cp` command to copy your files into the directory with two copy commands:

```
cp doc1 backup/doc1
cp doc2 backup/doc2
```

and `ls` will still say:

```
Directories:
backup newdirectory
Files:
doc1 doc2 f11e01 f11e02
```

If you list the contents of the subdirectory,

```
ls backup
```

you will see:

```
Files:
doc1 doc2
```

The files have been successfully copied.

### `chmod` — file protection and mode

As part of the directory entry, COHERENT keeps information about attributes of each file, including the time and date of creation or modification of the file. Also included is the *mode* of the file. It controls what can be done with the file and by whom.

For example, you can prevent other users from deleting, reading, or writing your files by setting the *protection* of the file. You can even prevent yourself from reading one of your own files, although this is not often done.

While there are many combinations of these attributes and different sets of users that they apply to, this document will cover only the basic combinations.

To change the protection or mode of a file, you will use the command `chmod` (meaning change mode). To make a file, say `doc1` in directory `backup` from the previous example, protected from writing or deleting, use the command:

```
chmod -w backup/doc1
```

where the `-w` means "remove write permission" and is followed by the file name.

To allow other users to read the backup file `doc2`, say:

```
chmod o+r backup/doc2
```

where the letter **o** signifies "other users", and the **r** tells `chmod` to grant read permission.

When files are created, they are set up with installation standard protection levels. Check with your system administrator or local documentation to be sure what the default protections are on your system.

To determine what the protection properties are for a given file, use the command

```
ls -l
```

The mode will be printed out as the first column for each file in the current directory. The format of the output from the `ls` command is described in the next section "Introduction to COHERENT commands".

### `rm, rmdir` — Removing files and directories

Files need to be removed to make way for new files. You may have old copies that you no longer need that are cluttering up your directory, or you may have accidentally created a file that you do not really want.

To remove a file, use the command `rm` (for remove). The parameter is the pathname of the file that you want to be removed:

```
rm backup/doc2
```

This example will remove the `doc2` backup that was created in an earlier example.

You can remove several files with a single command by listing them as consecutive parameters:

```
rm file01 file02
```

Files that have been protected as unwritable cannot be deleted. Suppose you created a file `tough` by typing

```
cat >tough
file1
file2
<ctrl-D>
```

and protected it by typing

```
chmod -w tough
```

If you try to delete the file with `rm`, the COHERENT system will type

```
tough: unwritable
```

This is done to prevent you from deleting a file unintentionally. If you do want to delete it, use the `-f` option for `rm`:

```
rm -f tough
```

and the file will be deleted.

You can also delete directories using the command `rmdir`. But before you delete any directory, it must be empty of files. Otherwise, you will get an error message, and the directory will not be deleted. The form of this command is

```
rmdir newdirectory
```

### `du, df` — How much space

You can determine how much disk space is taken up by your files with the command `du` (for disk used). This will tell you how many blocks are taken up by the files in your directory. If you have sub-directories, they will be listed separately.

Each *block* on disk contains 512 bytes or characters of information.

To determine how many blocks of information are available for use in the system, use the command `df` which shows you disk free blocks.

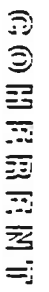
### `ln` — Linking files together

COHERENT has a unique feature that enables a file to have several names. These additional names are called *links*.

To create a link to an existing file, use the command `ln`. If you have a file named `doc1`, as you will if you have entered the previous examples, you can create a link to that file:

```
ln doc1 another
```

The protections and the data in the file will always be the same for both names `doc1` and `another`.



If one or the other of the names is deleted with the `rm` command, the data part of the file will remain. However, if both names are removed, then the data will also be removed. The data stays so long as there is at least one link remaining to the file.

### 5. Introduction to COHERENT commands

The commands that you enter into COHERENT are interpreted and acted upon by `sh`, a special COHERENT program called the *shell*.

This section shows some common commands in COHERENT. For more information on these or other commands see `help` and `man`. Also, consult the COHERENT Commands Manual.

You will need to be aware of some special punctuation characters. The special characters are:

```
* ? [ ] | : { }
( ) $ = : ' ' " < > << >>
```

Avoid these characters until you have read the Section 7 titled "More about COHERENT commands" which discusses their use, or until they are presented in examples.

#### `cat` — List contents of a file

A command that can be used to list the contents of a text file—a program's source, a document, or a message file—is `cat`. To list the contents of file `pgm` say

```
cat pgm
```

This will list the file on the terminal, using the *standard output*.

Another purpose for `cat`, and in fact the use from which it gets its name, is to concatenate several files on the standard output.

```
cat one two three
```

This command will list all three files `one`, `two`, and `three`, one after the other on the terminal. The files can be concatenated into another file by redirecting the standard output to the file. The special character '`>`' is used before the file name to indicate output redirection. The command

```
cat one two three >four
```

will concatenate files `one` `two` `three` into file `four`. `four` need not exist prior to this command, and if it does, the previous contents will be deleted.

>, <, >> — Redirection

When programs accept input from your terminal and write output to the terminal, they are doing so through files called *standard input* and *standard output* respectively. Much of the power of COHERENT stems from the fact that these files can be *redirected* to devices other than the terminal, or to COHERENT files. The redirection is signaled by '<' for standard input and '>' for standard output.

To illustrate, the COHERENT command `cat` will copy standard input to standard output if you say:

```
cat
one line
second line
<ctrl-D>
```

Try it. The lines that you type in following `cat` will be echoed back on your terminal. Since the I/O is buffered, the resulting output may not happen until you type the `<ctrl-D>`.

Redirect the standard output to a file by typing

```
cat >stuff
one line
second line
<ctrl-D>
```

The lines are not typed on your terminal, but are put in the file `stuff`. You can verify this by using `cat` to type the contents of the file:

```
cat stuff
```

Try this, and you will see the lines you typed in earlier appear on your terminal.

Since the COHERENT system treats devices, files, and programs uniformly, you can send the output from `cat` to the special file that is your terminal:

```
cat stuff >/dev/tty
```

This will act the same as

```
cat stuff
```

so long as the standard output has not been globally redirected, as is possible when commands are placed in a file. Commands in a file are discussed in section 7.

If you are directing standard output to a file, the file will be created if it does not already exist. If it does exist, the old contents will be deleted.

You can add new output to the end of an existing file rather than deleting the output by using a different form of output redirection:

```
cat >>stuff
line one
line two
<ctrl-D>
cat >>>stuff
line three
<ctrl-D>
cat stuff
```

The characters '>>' signify that output is to be added to the end of the file. The second `cat` command adds lines from the terminal to `stuff`. If file `stuff` does not exist, it is created.

Lower case sensitivity in commands

The commands shown in this manual are all in lower case characters. COHERENT treats upper case characters as distinct from their lower case equivalents. Therefore, the commands

```
cat
CAT
cat
cat
```

are all different, and only the last one is recognized by COHERENT.



## | — Pipes

As noted in an earlier section, the COHERENT pipe operator is used to build commands to do many things by combining building blocks of simple commands. The pipe connects the output of the command preceding it to the input of the command following it. The who command lists the users of the system, but in no particular order. If you want an alphabetical list, you can connect the sort command to the who command with a pipe:

```
who | sort
```

The output of who, normally directed to your terminal, will be directed to the input of sort, which normally gets its input from the terminal.

## scat — List files on screen

If the file you list with cat is more than twenty-four lines long, and your terminal is directly connected to the COHERENT computer, the beginning lines of the file will go quickly off the screen before you can read them.

At any point that COHERENT is printing on your terminal, you can cause it to halt temporarily by typing

```
<ctrl-S>
```

and the output will resume when you type

```
<ctrl-Q>
```

To be sure that you see all of the lines of the file output, use the scat command. When it has filled the screen with output, it will pause, waiting for you to hit <RETURN>. If you call scat with an option of -s,

```
scat -s file
```

blank lines will not be shown on your screen. With scat, you will not need to use <ctrl-S> and <ctrl-Q>.

## who — Who is on the system

To determine the user names of others who are currently using the system, use the COHERENT command who.

This command will list the names of those currently logged in to the COHERENT system, one name per line. You will see your own user name there as well.

If you find a terminal not in use that someone forgot to log out you can type a variant of the who command to see who the user of the terminal is:

```
who am I
```

This will reply with the name of the user of the terminal.

## ls, lc — Listing your directory

The previous section discussed two of the more commonly used commands: ls and lc. These will each list the files in a directory.

Presume that your directory has the files created in previous sections and that you did not remove newdirectory.

If you want to list files in your directory, simply use the command with no parameters:

```
ls
```

This will will produce

```
another
backup
doc1
doc2
file01
file02
newdirectory
stuff
```

lc lists file names like ls does, but in columns across the screen with files and directories listed separately.

```
lc
```

will give:

Directories:  
backup newdirectory

Files:  
another doc1 doc2 file01 file02  
stuff

If you want to list files in a directory other than your own, specify the name of that directory as a parameter to the command. For example, /bin is a directory in the COHERENT system that contains commands. Type

```
lc /bin
```

You can specify options to each of these directory listing routines. To do so, precede the option with a hyphen (and no intervening space). The option must appear before any other parameters. To list only the files in the directory for user carol, leaving out any directories, use the f option with lc:

```
lc -f /usr/carol
```

or, if you type the command

```
lc -f
```

for your directory, the COHERENT system will reply

```
Files: doc1 doc2 file01 file02
```

The commonly used options for lc are:

```
-d      List directories only, omitting files
-f      List files only, omitting directories
-l      List files one per line, not in columns
```

The ls command produces a list of file names, one per line, and optionally much more information. To produce all the information, use the l option:

```
ls -l
```

A sample output of the long listing produced by this option is shown here:

|            | mode | # owner | size<br>bytes | In  | modification<br>date | time  | name         |
|------------|------|---------|---------------|-----|----------------------|-------|--------------|
| -rw-r--r-- | 1    | you     | 17            | Wed | Aug 19               | 17:51 | file01       |
| drwxrwxrwx | 2    | you     | 32            | Wed | Aug 19               | 17:53 | backup       |
| -rw-r--r-- | 1    | you     | 17            | Wed | Aug 19               | 17:52 | doc1         |
| -rw-r--r-- | 1    | you     | 17            | Wed | Aug 19               | 17:52 | doc2         |
| -rw-r--r-- | 1    | you     | 17            | Sat | Aug 15               | 17:20 | file01       |
| drwxrwxrwx | 2    | you     | 32            | Wed | Aug 19               | 17:52 | newdirectory |

Headings have been added here to show the meaning of each column.

The mode column is made up of four separate subfields. This field describes the access permissions for the file and whether or not the file is a directory. Taking the entry for file file01 as an example, we have:

```
-rw-r--r-- 1 you          17 Sat Aug 15 17:20 file01
|  \  \  \  \
|  |  |  |  |
|  |  |  |  |
|  2  3  4
```

The leftmost position has been labeled 1. If the file is a directory, this position will contain a d, otherwise it will contain a hyphen.

The remainder of the mode field is three subfields, each with three characters. Subfields 2 through 4 contain three positions each. These fields represent permissions to be granted to different groups of users.

Subfield 2 is for the owner of the file. Subfield 3 is for members of the group that the owner is in, while subfield 4 is for all other users. The topic of groups is not covered in this manual.

The three positions within each of these subfields represents the permissions to read, write or execute the file:

```
rwX
```

If the permission is to be granted, the corresponding letter is printed. A hyphen indicates that the permission is denied.

The *read* permission means that the file can be read, for example by *cat*, if the permission is granted. If *write* permission is granted the file can be written to, as well as deleted. The *execute* permission signifies that the file contains a command and can be executed.

The column labeled *#* represents the number of *links* to the file for non-directory files.

The column labeled *owner* names the user who owns the file. You will usually be the owner of files in your directory.

*Size* shows the number of bytes used in the file.

Next is the *date* and *time* that the file was last modified, for example, by *ed*.

Finally, the *name* of the file is shown.

#### `msg` — Send message

You can send a short message to a user currently logged in to the system by using the `COHERENT` command `msg`. To illustrate, send a message to yourself. Type:

```
msg you
this is a test message
```

substituting your user name for 'you' in the `msg` command. The result will be:

```
you: this is a test message
```

#### `msg` — Hear no messages

If you do not wish to receive online messages, the command `msg` will prevent other users from interrupting your work:

```
msg n
msg y
```

Later, you can allow messages again by saying

To determine which of the two `msg` options is in effect, use the `msg` command with no option:

```
msg
and it will tell you what the current setting is.
```

#### `write` — Electronic discussion

To carry on a more extended dialogue, the command `write` is more convenient than repeatedly issuing `msg` commands. Once started, the dialogue will continue until a `<ctrl-D>` is typed on the terminal.

To begin the communication using `write`, `jack` will issue a command

```
write jill
```

indicating communication with user `jill`. On `jill`'s terminal, the message

```
Message from jack...
```

will appear. To establish the other half of the communication, `jill` should then say

```
write jack
```

and a similar notification will appear on `jack`'s terminal.

At this point, both users simply type lines on their terminal and `write` sends the message to the other user. To avoid typing at the same time as the user you are communicating with, it is recommended that each message be ended with a line having the single letter

```
o
```

signifying "over", or "go ahead". When the other user sends you this, you know it is your turn to send a message, and vice versa.

When your communication is finished, you should type

```
oo
<ctrl-D>
```

Here, `oo` means "over and out", and the `<ctrl-D>` will exit the `write` command.

**mail — Mailing a letter**

If a user is not logged in to COHERENT, or you want to send a less immediate note, use the command **mail**.

To send mail to jill, say:

```
mail jill
```

and type the mail message beginning on the following line. You can also mail messages that have been previously put into a file. For example,

```
ed
a
All come to the birthday party at four
next to the pump room.
w hb.msg
q.
```

Now mail the message by typing

```
mail jill <hb.msg
```

If the mail is coming from the terminal, terminate the message with a **<ctrl-D>** or a line containing only a period.

You can send a mail message to several users at one time by listing their user names on the command line:

```
mail jill Jack ted barb <hb-party
```

will send the message **hb.party** to jill, Jack, ted and barb. To illustrate the use of the mail command, send yourself a mail message by:

```
mail you
This is a note to
myself to test
mail.
```

and substitute your user name for "you" in the mail command.

If someone has sent you mail, the COHERENT system will tell you:

You have mail.

when you log in.

To receive mail, type the mail command with no parameters:

```
mail
```

If you have no mail, COHERENT will tell you

```
No mail.
```

If you do have mail, each message will be typed out on your terminal along with the user name of the sender, and the date and time that the mail message was sent.

After each message, the mail program types a question mark **?** and waits for your reply. Give a **d** if you wish to delete the message that you have just read, a **<RETURN>** to go onto the next message without deleting the message you just read, a **s** command to save the mail message in the file **mbox**, or the command **q** to exit the mail program.

You will know that you are finished with all of your messages when mail scuds you a **?** without typing anything before it.

**pr, lpr — Print files**

Earlier in this section, the **scat** command was presented as a way to list files on your terminal. However, for hard copy or higher-volume output, the line printer is more convenient. The command **lpr** will print files for you, making sure that your request does not conflict with other uses of the printer.

To print a file, issue the command

```
lpr file
```

substituting the name of the file to be printed for *file*. If you want a banner on the first page of output, use the **-b** option:

```
lpr -b banner file
```

If no file is given, the standard input is printed. Thus, **lpr** can be used in pipes.

No processing of the file is done by `pr`. This means that no page headings are printed. Another command `pr` will do this for you. It will paginate the standard input, giving a header with date, file name, page number and line numbers. The paginated output appears on the standard output.

To print a paginated file on the line printer, say

```
pr file | lpr -b hammer
```

`echo` — Echo the command line

The `echo` command will type on the standard output the value of its parameters. For example, the command

```
echo five six
```

will type out on the terminal

```
five six
```

While this may not seem to do very much, it can help you find out exactly what the parameters to any command will be if there are any special characters involved.

For example, if you had problems with a command of the form

```
cat **
```

and you wanted to be sure what the parameters were going to be, you should give the same parameter string to `echo`:

```
echo **
```

and you will see how the parameters have been transformed. In this case, the parameters will be a list of file names in the current directory. To be sure that the double asterisk itself is used as a parameter, enclose it in single quotes:

```
echo '***'
```

and the result will be

```
***
```

on the terminal.

#### `ed` — Text file editor

There are many uses for files on the COHERENT system—user manuals, notes, source programs, mail, and so on.

To create or change text files, use the COHERENT command `ed`. This calls the text file editor program which is a powerful yet easy to use interactive text editor.

With `ed` you can create files interactively, add more text to files, rearrange paragraphs in a file, and correct spelling errors.

For a full description of all the `ed` features, see the *ed Interactive Editor Tutorial*.

#### `grep`—Find patterns in text files

`grep` gives you the power to find lines containing a *pattern* in one or more files. Patterns are sometimes called *regular expressions*.

To illustrate `grep`, create file `doc1` by typing:

```
cat >doc1
a few lines
of text.
<ctrl-D>
```

Then the command

```
grep text doc1
```

will print out the second line of file `doc1`:

```
of text.
```

The first parameter to `grep` is the *pattern* that you are looking for, and the rest of the arguments are the names of files to be examined. Text is the pattern and `doc1` is the file.

To find out if a particular user is on the system, pipe `who` into `grep`:

```
who | grep you
```

substituting the user name in question for you. Try it with your user name. The pattern is you, but there is no file name specified. `grep` will read input from the standard input file, which in this example is connected to the output of the `who` command.

You can specify several files to be searched. Simply put the additional file names after the first:

```
grep pattern doc1 doc2
```

Or, you can search all files in the current directory for the pattern with

```
grep pattern *
```

The \* will be interpreted to mean all files, and grep will look for *pattern* there.

The search pattern can be a *pattern*. Patterns are fully discussed in the *ed Interactive Editor Tutorial*. The name `grep` is derived from the `ed` command

```
g/re/p
```

where "re" means regular expression, or pattern. In giving a pattern to `grep`, be sure that you enclose it in single quotes. Otherwise, the shell will interpret the pattern expression before `grep` sees it.

You can also locate lines that do *not* contain given patterns by using the `grep` option `--v`.

```
grep -v bugs prog1 prog2
```

This command will find and print all lines in files `prog1` and `prog2` that do not contain bugs.

**date** — Print the date

The COHERENT system keeps track of the time and date. To find the date and time, use the command

```
date
```

and COHERENT will respond with the day of the week, the month and year, and the time of day.

Internally, the date and time is kept in seconds since January 1, 1970 at 00:00:00 GMT. This means that files created in one time zone and referenced in another time zone will bear the correct time. The time and date printed out is converted from the internal form to the local time.

**time** — Measure command execution time

You can measure how long any command will take with the `time` command. This can be useful if you are doing improvements to a program and need to measure the time it takes, or are determining how long a program takes under different conditions of input data.

To use the `time` command, precede the command that you are timing with the `time` keyword. For example, to time how long it takes to list the users on the system, type

```
ttime who >temp
```

and when the `who` command is finished, the `time` command will print out the amount of time the command took, the amount of time spent in the `who` program, and the amount of time spent in COHERENT itself. The result will look somewhat (but not exactly) like:

```
Real: 0.9
User: 0.1
Sys: 0.2
```

This command will give different results depending on the size of your computer and the number of users on it when you type the command. The `Real` number (0.9) is the amount of elapsed time taken by the command. The `User` time (0.1) is the amount of time spent in the command `who` itself, and the `Sys` time (0.2) is the amount of time that COHERENT itself spent processing the job.

**passwd** — Passwords

You may wish to change your password from time to time for greater security.

Changing passwords on the COHERENT system is easy.

Type the command `passwd`, which will first ask you for your current password (if you have one), and then ask you to enter your new password twice. Entering the new password two times helps ensure that the system gets the password as you want it. If you do not type it the same way both times, COHERENT will say

```
Password not changed.
```

and you must begin again with the command `passwd`.

Be sure the password is something that you can remember. It is recommended that the password be at least six characters long. Do not write it down, but commit it to memory. You can use a four-letter password, but if you do, you should mix upper case and lower case letters to increase the secrecy of the password.

#### stly — Changing terminal behavior

Because there is a wide variety of terminals used with the COHERENT system, even of the video type, it is necessary for the COHERENT system to know certain things about your terminal.

The command

```
stly
```

will describe the information COHERENT currently has for you, and you can tell COHERENT to treat your terminal differently with the command `stly` with parameters.

Normally, each character you type is echoed by the COHERENT system. This means that when you type a character, the system types it back to you so that it appears on your screen. If you have a terminal that is also echoing the character, you will see double characters. To prevent this, issue the command

```
stly -echo
```

You can also use this if your terminal is not echoing the characters, but you will be typing in the dark.

To set the echo feature again, say

```
stly echo
```

When you first log in, the system presumes that your terminal does not directly handle the tab character, so when COHERENT sends a tab to your terminal, it simulates it with spaces. If your terminal does handle tabs, issue the command

```
stly tabs
```

and the COHERENT system will no longer substitute spaces for tabs. To go back to substitution,

```
stly -tabs
```

The `<ERASE>` character allows you to delete the previous character. The `<KILL>` character allows you to delete the line that you have been typing but have not yet finished. You can change these as you wish with commands of the form

```
stly erase "E KILL" K
```

This particular example will set the erase character to `<ctrl-E>` and the kill character to `<ctrl-K>`. The up-arrow or caret character `^` tells `stly` that you want to specify a control character.

To reset erase and kill to their values at login, the command

```
stly ek
```

will suffice. This is equivalent to

```
stly erase # KILL @
```

To determine what your current terminal parameter settings are, type

```
stly
```

with no parameters. The command will show you the current settings of all the options.

6. Miscellaneous tools and features

This section describes several useful CURRENT commands in detail.

units — Convert units of measure

CURRENT provides a program to convert from different units of measurement. The program units has an encyclopedic knowledge of units of measure.

To use the program, enter

units

After a short delay, units will ask you

You have:

to which you reply with the unit to be converted from, say cm. Then, units will ask

You want:

Reply inches. The entire dialogue thus far will appear:

You have: cm  
You want: inches  
\* 0.3937  
/ 2.54000508001016

This means you should multiply the centimeters value by 0.3937 (prefixed by \*) to get inches, or divide the inches value by 2.54000508001016 (prefixed by /) to get centimeters.

You have: 98 cm  
You want: inches  
\* 38.5826  
/ 0.0259184191837714

which tells you that there are approximately 38.6 inches in 98 centimeters or that there are 0.259184191837714 of 98 centimeter units in one inch.

You can also combine units, such as "miles per hour". To convert from a common measure of velocity to one less frequently seen, you can say:



```

You have: 60 miles/hr
You want: furlongs/fortnight
      * 161280
      / 6.200396825396825e-06

```

This tells you that if you traveled at a steady rate of 60 miles per hour, you would cover 161,280 furlongs in one fortnight. Notice that the second number prefixed by / ends with "e 06". This is scientific notation meaning "10 to the minus sixth power".

The number of units included in the program is considerable—currently 800. It is possible, although quite unlikely, that you will come up with a unit that this program does not know about.

Notice that units performs multiplicative conversions. It will not perform conversions requiring addition or subtraction, as in Fahrenheit to Kelvin.

#### bc — Desk calculator

Another handy tool is the COHERENT desk calculator program bc. It is like having a powerful calculator at your fingertips.

```

If you type
bc
2 + 2
bc will reply
4

```

You can adjust the number of positions held to the right of the decimal point by a statement of the form

```
scale = 13
```

This makes bc carry 13 decimal positions.

The number of positions to the left of the decimal point depends upon the calculation requirements, and is automatically expanded by bc to prevent overflow. The number of digits carried is limited only by the amount of available computer memory. For example, try

```
2^500
```

The result will be

```

3273390507896141870013189696827599152216\
6420460430647894832913680961337964066745\
5488327009232590415715088668412756007100\
9217256545885393053328527589376

```

You do not need a print statement. When bc sees any formula like "2 + 2" or a number like "3777", it will print the result on your terminal.

bc understands the following elementary operations within formulas:

```

+      add two numbers
-      subtract the second number from the first
*      multiply the two numbers
/      divide the first number by the second
%      remainder of division of first by second
^      first number raised to power of second
quit  exit the bc program

```

Each of these operators appears between two names or numbers. Names are like variables in formulas.

You can add comments to your bc programs by enclosing them in /\* and \*/:

```
n=10 /* Initial value of a */
```

bc has several special operators that apply to single names. If you give to be the input

```

b=30
a=20
r=a+b

```

the special operator + will change the value of a by adding one to it and will use the new value to add to b. The number printed for this example will be 51 (try it), and the value of a will be 21.

The special operators are illustrated by the following example program:

```

a++      /* add one to a      */ /* */
a--      /* subtract one from a */ /* */
a+=2     /* add two to a and store in a */ /* */
b+=a     /* b becomes b plus value of a */ /* */
b-=a     /* b becomes b minus a      */ /* */
c*=b     /* c becomes c multiplied by b */ /* */
c/=a     /* c becomes c divided by a   */ /* */
c%=b     /* c becomes remainder of c divided by b */ /* */
d*=3     /* d becomes d raised to the 3rd power */ /* */

```

Notice that `++` and `--` can be used before or after a name. When used in front of a name, the name takes on the new value, and the result of the operation is the new value. The input

```

a = 10
++a

```

will print the result 11 and the new value of the name `a` will be 11.

When `++` and `--` are used following a name, the value used in the expression will be the old value of the name, but the name will take on the new incremented or decremented value. The statements

```

b=20
b--

```

will print 20 (the old value of `b`), but `b` will take on the new value of 19.

The operations in the last two examples are called pre-incrementing and post-decrementing, respectively. The notation

```
a += b
```

is shorthand for

```
a = a + b
```

and means exactly the same thing. To square the current value of `s`, type

```
s *= 2
```

remembering that `*`, or `pow`, signifies the *power* operation. This is equivalent to typing

```
s = s * 2
```

or

```
s = s * s
```

In the `bc` examples shown so far, all names have been one letter. Names in fact may be unlimited in length. Names need not be declared before use, and if they are used before an assignment is made to them, are presumed to have the value zero. However, it is good programming practice to explicitly initialize all variables used.

The statements shown so far have been either assignment statements, giving a new value to a name; or a formula, which prints out the resultant value. Several other kinds of statements are available.

An example of the `if` statement will print the value of `x` if it is greater than 200:

```
if (>200) x
```

The while statement will repeatedly execute statement so long as a decision expression is true. The statements

```

i = 0
while (++i <= 10) i

```

will print integers one through ten. Try it!

The `for` statement is a bit more complex and will not be discussed in detail here. It resembles the construct of the same name in the C programming language (as do all the statements in `bc`). For a discussion of how the `for` statement works, and a complete discussion of `bc`, see the *bc Desk Calculator Tutorial*.

The first of these statements, the `if` statement, means "execute a statement if and only if a decision expression is true". In this statement and the `while` statement, the expression `E` can include the following relational operators:

```

==      two operands equal
!=      two operands unequal
<=     first operand less than or equal to second
<      first operand less than second
>=     first operand greater than or equal to second
>      first operand greater than second

```

The statement `quit` causes `bc` to finish processing and to return to the `COHFERENT` system.

To describe the statements formally:

```

IF ( E ) S
while ( E ) S
for ( E ; E ; E ) S
{S; S; ... ; S}
break
quit

```

Here, the letter *S* means statement, and the letter *E* means expression. Thus, an `if` statement may have another `if` statement as part of it.

Finally, you can define functions in `bc` using the special keywords `define`, `auto`, and `return`:

```

define n (p1, ... , pn)
{
    ...
}

```

where *n* is the function name, *p1* through *pn* are the parameters or arguments to the function, and the braces `{` and `}` enclose the body of the function definition.

One of two statements used only in functions is the `auto` statement:

```

auto n1, n2, ... , nn

```

It defines names *n1* through *nn* to be automatically allocated by `bc` when the function is called. These names are separate from any names outside the function and are separate for each use of the function, even if it calls itself.

An example of a function definition is for Fibonacci numbers:

```

define fib (r) {
    if (r==0) return (0)
    if (r==1) return (1)
    if (r > 1) return (fib (r-1) + fib (r-2))
}

```

To call the function and print the result, say

```

fib (5)

```

The `COHFERENT` system command to call `bc` can have the file name of a `bc` program:

```

bc fib.bc

```

This causes `bc` to begin by reading the program in `fib.bc`. Use `ed` to enter the above function definition into the file `fib.bc` and try this.

After `bc` has read the file, it will then read from your terminal. This capability will allow you to put the definition of `fib` into `fib.bc` using `ed` and call the function `fib` from your keyboard. You can put more than one file on the `bc` command line to enable you to use several predefined `bc` programs at once.

For more information on `bc`, see the *bc Desk Calculator Tutorial*.

`cal` — *Calendar maker*

You can produce a calendar for the year on your hard-copy terminal or printer with the command `cal`. In fact, you can print one for (almost) any year that you choose. Simply say:

```

cal 1976

```

to get the calendar for the United States' Bicentennial year, or

```

cal 1981

```

for the year in which this document was written. For earlier dates, beware of the change to the Gregorian calendar, since not all countries changed when England did in 1752.

You can produce the calendar for a single month by specifying month in addition to the year. To see an unusual month, type:

```
cal 9 1752
```

### crypt — Secure information

COHERENT provides tools that allow you to process information securely. The encryption program `crypt` will perform a secure encryption of a file. To encode the file `secrets` using a key `alpha` use the command:

```
crypt alpha <secrets >encoded
```

and the file will be encrypted into the file named `encoded`. To decrypt the file, use the same key on the encoded file:

```
crypt alpha <encoded >decoded
```

The file decoded will contain the same information as the original file. You can use any key—just don't forget what it is.

## 7. More about COHERENT commands

COHERENT commands are read and acted upon by the *shell*. This program provides a great range of commands, from the relatively simple ones presented in earlier examples, to complex command programs involving variables and control constructs. Commands can return values, which enable following commands to execute conditionally. These and other features enable you to construct command programs and save them in a *script* file that is easy for you or another COHERENT user to call upon, yet performs a complex sequence of steps.

### Simple commands

The shell command language is built around simple commands. Many have been shown in examples already, such as the command to list your directory:

```
ls
```

Several simple commands may be combined on one line by separating them with semicolons:

```
who;du;mail
```

The commands are executed in sequence as if they had been typed

```
who
du
mail
```

In both of these examples, `du` will not begin execution until `who` is finished, and `mail` will not begin until `du` is done.

### Special characters

If you are familiar with `ed`, you know that there are certain characters that have special meaning to `ed` and are used with care.

The shell also treats certain characters specially, and therefore if you want to use them without their special meaning, you must precede them with the backslash character `\`, or enclose them in quotes.

```
* ? [ ] | ; { } ( )
$ = : . ! " < > << >>
```

The function of these characters will be explained later in this section. To use one of these characters in a command, for example '?', you will type

```
echo A?
```

Additionally, the shell treats certain words in a special way when they appear as the first word of a command:

```
case do done elif else esac
fi if in then until while
```

If you need to use one of these as the first word of a command and you do not want the shell to recognize the special meaning, then enclose the word in single quotes "'":

```
'if'
```

### Background commands

Commands can be executed simultaneously rather than sequentially by the shell. If a command is followed by the special character '&', the shell will begin executing it immediately, and will prompt you for another command. If you need to *sort* a large file, but want to continue with other commands while the *sort* is taking place, you can type:

```
sort >staff.sorted staff.unsorted &
ed prog
```

and edit file *prog* at the same time.

When you run a command with &, the shell will type the *process id* of the command started in background. Each running command or program in the COHERENT system is assigned a *process id* when it begins executing. Normally, there is no need to be concerned about these numbers. But when you run background commands, the shell tells you the process id of the background so that you can keep track of its execution.

The command

```
ps
```

will list the processes you are currently running. If you have no background jobs, the response will be

```
TTY PID
30: 362 -sh
30: 399 ps
```

The first column shows the number that COHERENT has internally assigned to your terminal. This is the same terminal number printed out by *who*. The second column shows the process id; the third column shows the program or command executing. The characters — *sh* in the third column means the shell. There are two processes because the shell is running the *ps* command as a separate process.

Once you have started a background command, the *ps* command will show you the process entry, which will have the process *id* that the shell typed out for you.

If you need the results from a background job, you can wait for it to finish by issuing the command

```
wait
```

The shell will then accept no further commands until all your background jobs are finished. If there are no background jobs, there will be no delay.

### Commands in a file

Many of the commands that you use in COHERENT are programs, such as *ed*. Others, like the *man* command, are files containing still more commands. You don't need to know which of the two commands you use.

You can build files containing commands. If you have a frequently-used set of commands, you can include them in a file and save on your typing.

For example, assume that you wish to periodically check the amount of disk space that you have used, the amount of disk space still available, and examine what users are on the system. Build a

file named `good.am` of commands by typing the following information:

```
ed
n
dn
df
who | sort
mall
w good.am
q
```

To call up these commands, you need only say

```
sh good.am
```

where `sh` is a command that means: read commands from a file, in this case `good.am`. Any commands you issue from your terminal you can do from a command file, or *script* as this is called. The command file can also be created as

```
du; df; who|sort; mall
chmod +x good.am
```

will enable you to execute the script `good.am` by typing

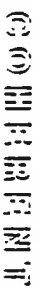
```
good.am
```

and leaving off the `sh`. Once you have done the `chmod` command, you can still issue the commands by

```
sh good.am
```

as well as use `ed` to change the contents of the file.

Notice that the commands called by a script may themselves be scripts. This is illustrated by the script `second.sh`.



```
ed
n
sh good.am
to
w second.sh
q
```

so that issuing

```
sh second.sh
```

calls the script `good.am`, which calls the command `to`.

#### `.profile` — Login shell file

Once you are logged into the system and before you are issued your first prompt, COHERENT checks your home directory for a file named `.profile`, and if it is present, reads the commands it contains.

This enables you to have COHERENT execute commands as soon as you log in. Check to see if your installation provides one for you by doing `an le` (be sure that your current directory is the home directory). If the file is there, type it out by saying

```
cat .profile
```

Some of the commands may be of the form

```
PATH=':/bin:/usr/bin'
```

This sort of command will be discussed below.

#### Substitutions

Scripts of the form shown above are processed by the COHERENT shell without change.

The COHERENT shell increases the power of commands by performing three kinds of substitutions within commands before they are executed.

First, special characters in commands will be replaced by file names from the current or other directories. With this capability you can issue a single command that will process several files.



Secondly, you can give a script parameters, much like parameters that are passed to a Pascal, Algol, or C procedure. This allows you to target the action of the script to a specific file name specified when you call it.

Thirdly, the output of one command can be substituted into another command. This gives a great amount of power to your command file usage.

The `echo` command will be used to illustrate the effects of these kinds of substitutions. Remember that substitutions take place for all commands in the same way that they do for the `echo` command.

#### File name substitution

Many command parameters are file names.

By using special shell characters, you can substitute file names in commands. These special characters describe the file name *patterns* for the shell to look for in the directory. When the file names are found, they replace the pattern.

The pattern character `*` will match any number of any characters in file names. Thus,

```
echo *
```

will echo all the file names in the current directory, while

```
echo f*
```

will give all file names that begin with the letter `f`, and

```
echo a**z
```

will list all names that begin with `a` and end with `z`.

To illustrate more clearly, create two files by typing

```
cat >zz1
<ctrl-D>
cat >zz2
<ctrl-D>
```

Then the `echo` command

```
echo zz*
```

will produce the output

```
zz1 zz2
```

Thus, by using a single `*`, you can substitute several file names into a command. In other words, the command

```
echo zz*
```

is equivalent to

```
echo zz1 zz2
```

If there are no file names that fit the pattern, the special characters are not changed, but left in the command exactly as you typed them. To illustrate, type the command

```
rm zz*
echo zz*
```

The first command will remove all files whose names begin with `zz`, and is therefore equivalent to

```
rm zz1 zz2
```

The `echo` command that follows, however, will echo

```
zz*
```

because there are no files beginning with `zz`; they were just removed.

Enclosing command words with the single quote character `'...'` will prevent the shell from matching file names with the enclosed characters. In the unlikely event that you have a file whose name is

```
zz*
```

that you want to remove, use the command

```
rm 'zz*'
```

The `*` is enclosed in single quotes, and will therefore not be changed by the shell.

Another special character `?` will match any single letter. Create empty files `file1`, `file2`, and `file3` by typing:

```
>f11e1
>f11e2
>f11e33
```

then the command

```
echo f11e?
```

will reply with

```
f11e1 f11e2
```

since ? will not match 33.

The bracket characters [ and ] can be used to indicate a choice of single characters in a pattern:

```
echo f11e[12]
```

This command will reply

```
f11e1 f11e2
```

To match a range of characters, separate the beginning and end of the range with a hyphen. The command

```
echo [a-m]*
```

will print any file name beginning with a lower case letter from the first half of the alphabet, and is exactly equivalent to

```
echo [abcdeghjklm]
```

When such patterns find several file names, they are substituted in this manner, they will be inserted in alphabetical order.

Since the character / is important in file pathnames, it is not matched by \* or ? in patterns. The slash must be matched explicitly, that is, it will only be matched by a / itself. Therefore to find all the files in the /usr directories with the name notes, type:

```
echo /usr/*/notes
```

The asterisk will match all the subdirectories of /usr that contain a file named notes. Additionally, a leading period in a filename must be matched explicitly. If you have a file in your current directory with the name .profile, the command

```
echo *f11e
will not match it.
```

These patterns can appear anywhere within a command or a command file.

#### Parameter substitution

Each shell script can have up to nine positional parameters. This enables you to write scripts that can be used for many circumstances. Recall that command parameters follow the command itself and are separated by tabs or spaces. An example of a command reference with two parameters is:

```
show first second
```

where first and second are the parameters.

To substitute the positional parameters in the script, use the character \$ followed by the decimal number of the parameter. For example, build the script show by typing

```
ed
n
cat $1
cat $2
diff $1 $2
w show
q
chmod fx show
```

The \$1 and \$2 refer to the first and second parameters respectively. Create two sample files:



```

cat >first
line 1
line two
line 3
<ctrl-D>
cat >second
line 1
line 2
line 3
<ctrl-D>

```

Then, issue the show command

```
show first second
```

which has the same effect as typing

```

cat first
cat second
diff first second

```

If you issue the show command with fewer than the required number of parameters, the shell will substitute an empty string in its place. For example, using the command with only one parameter

```
show first
```

is equivalent to

```

cat first
cat
diff first

```

where the null string has been substituted for \$2.

The example above shows the parameter references separated from each other by a space. In some uses, you may wish to prefix a substituted parameter to a name or a number. When more than one digit follows a \$, the shell picks up the first digit as the number of the parameter. To illustrate, build a shell file pos:

```

ed
a
echo $167
.
w pos
q
chmod +x pos

```

Then call the script with

```
pos five
```

and the result will be

```
five67
```

#### Shell variable substitution

In addition to positional parameters, the shell provides shell *variables*. Variables can be assigned values, tested, and substituted in commands.

The variable name can be constructed from letters, numbers and the underscore character. Sample names are:

```

high-tension
a
directory
167

```

Note that keywords must not be single digits, because the shell will treat them as positional parameters. Be aware that upper case letters and lower case letters are treated differently in shell variable names.

Values are given to shell variables by an assignment statement:

```
a=welcome
```

You can inspect their value with the echo command

```
echo $a
```

The value of the variable **a** is substituted in the echo command, which then appears as

```
echo welcome
```

CORIENT will respond to this command by typing

```
welcome
```

Don't forget the \$ when referring to the value.

Notice that the shell will be looking for special characters in any command that it sees. This includes the space character. To avoid problems, enclose the value to be assigned in single quotes:

```
phrase='several words long'
```

There are several uses for variables. One is to hold a long string that you expect to type repeatedly as part of a command. If you are editing files in a subdirectory like

```
/usr/wisdom/source/wldget
```

you can abbreviate if you set a variable pw to

```
pw='/usr/wisdom/source/wldget'
```

Then simply using \$pw in a command

```
echo $pw
```

will substitute the long pathname.

Another use of shell variables is for keyword parameters to commands. These then can be used the same way that positional parameters are. Create another script resembling show:

```
ed
n
cat $one
cat $two
diff $one $two
*
w show2
q
chmod +x show2
```

To use show2, issue

```
one=first two=second show2
```

which will be equivalent in effect to

```
cat first
cat second
diff first second
```

Unlike positional parameters, keyword parameters may be several characters in length. If you want some text immediately to follow a keyword parameter, enclose the keyword parameter in braces. Build a command file brace to illustrate:

```
ed
n
echo 'with brace: ' ${n}bc
echo 'without brace: ' $nbc
*
w brace
q
chmod +x brace
```

Call the command file with a set:

```
a=567 brace
```

The result will be:

```
with brace: 567bc
without brace:
```

When used in this way, the keyword parameters must be assigned before the command and on the same line as the command. In this case, the assignment of keyword parameters does not affect the variable after the command is executed. For example, if you type

```
one=ordlnl
one=first two=second show2
echo 'value of one is ' $one

the echo command will produce

value of one is ordlnl
```

Variables set other than on the line of a command are not normally accessible to a script. To illustrate, build a parameter display script:

```
ed
n
echo 1 $1 2 $2 p1 $p1 p2 $p2
w pars
q
chmod 4x pars
```

This will be used to show the behavior of parameters. The parameters to echo without a \$ will help to read the output. To pass positional parameters, type

```
pars ay bee
```

and the output will be

```
1 ay 2 bee p1 p2
```

To pass keyword parameters, type

```
p1=start p2=begin pars
```

and the result will be

```
1 2 p1 start p2 begin
```

To illustrate that the setting of p1 and p2 did not "stick", type

```
echo $p1 $p2 'to show'
```

and echo replies

```
to show
```

indicating that p1 and p2 are not set.

Illustrating that variables set separately from a command are not seen by the command, type

```
p1=outside1 p2=outside2
pars
```

which will reply

```
1 2 p1 p2
```

By using the export command, however, such variables can be made available to commands. The commands

```
export p1 p2
p1='see me' p2=hello
pars
will reply
```

```
1 2 p1 see me p2 hello
```

thus indicating that after the export of p1 and p2 they are available to other commands. Once a variable has appeared in an export command, its value can be changed without a need to export it again.

#### Command substitution

By enclosing a command in ` characters, you can feed its output into another command. For example,

```
echo `ls`
```

will echo the output of the ls command.

This can be a handy way to generate parameters for a command from a prepared file. Assume the file chapters contains a list of file names of chapters.

```
ed
n
ch1
ch2
apndxxA
apndxxB
w chapters
q
```

Create a sample script scr1

```

ed
a
for 1
do cat $1
done
*
w scr1
q
chmod +x scr1

```

These can be passed as parameters to a script file `scr1` by

```
scr1 `cat chapters`
```

and each of the files will be processed with `cat`.

### Special shell variables

When you log in to the COHERENT system, the shell variable `HOME` is set to your *home* or default directory path. If your user name is `henry`, then the command

```
echo $HOME
```

will reply

```
/usr/henry
```

on many systems. The change directory command `cd` sets the working directory to the path found in `HOME`; if no argument is given.

The shell normally prompts you with `$` for commands, and with `>` if more information is needed. These two prompts are taken by the shell from the variables `PS1` and `PS2`. You can change these if you want different prompts, for example

```
PS1=#
PS2=#: ;
```

To make these have effect each time you log in, put the assignment statements in your `.profile` file.

The shell variable `PATH` contains a list of pathnames of directories containing commands. The contents of `PATH` shown by typing

```
echo $PATH
```

is typically

```
:/bin:/usr/bin
```

This means that the shell will look for the command first in the current directory, then in `/bin` and, if not found there, then in `/usr/bin`. The pathnames are separated by `:`. This means that there is an empty string preceding the first `:`, the current directory. Another common setting for `PATH` is

```
...:/bin:/usr/bin
```

meaning that commands are first sought in the current directory, then in `.`, the parent directory to the current directory, then in `/bin`, and finally in `/usr/bin`.

### `.*` — Read commands

Similar to the command `sh` is the `.` command. The command

```
./cfl1
```

causes the shell to read and execute commands from `cfl1`.

This is different from the `sh` command in several respects. First, there isn't any way to pass parameters to `cfl1` with the `.*` command. Second, the `sh` command executes another shell to read the commands, while `.*` simply reads the commands directly. And finally, all the string variables and parameters are accessible by `cfl1`.

The command file `good.am` created earlier can be executed with

```
./good.am
```

and will have the same effect. Similarly, the `.*` can itself be used within a command file:

```
ed
a
./good.am
!c
*
w th1rd.sh
q
```

Then, the command

lhr1.sh  
will have the same result as the command

sh lhr1.sh  
Most COHERENT commands return a value indicating success or failure. For example, if Pascal cannot find your source file, it will issue a diagnostic, as well as return a value. You can examine this value by typing the command

echo \$?

This will tell you the value returned by the last command executed. Commands that return a failure value usually also type a message indicating the error condition. The value zero indicates success or truth, while a non-zero value indicates failure or falsehood.

You can use the value returned by commands to effect decisions about executing other commands. To illustrate, the cmp command with an option of -s will not print differences, but only return the value, which is zero if the files are identical, and one if the files are different.

test — Condition testing

For most commands the return value is a side-effect of their operation. However, the test command's only task is to return a value. Many conditions may be tested with this command.

To determine if a file exists, the command

test -f f11e01

will return true if file1 exists and is not a directory. To check if a file is a directory, use

test -d f11e01

Strings can also be examined by test. This is useful when parameter substitution is used. To illustrate, build the following command:

cd

a

test \$1 = \$2

echo 'test 1 & 2 for equal: ' \$?

test \$1 != \$2

echo 'test 1 & 2 for not equal: ' \$?

.

w test.sh

q

chmod +x test.sh

Since it is a parameter, be sure that the "" = "" in the test command is preceded and followed with a space.

This command file will test the two parameters for equality. Try the commands

test.sh one two  
test.sh one one

There are other options for the test command. To see them all, type

man test

Conditional command processing

Type the following commands to create two files:

cat >f11e1

line one

line two

line three

<ctrl-D>

cat >f11e2

line one

two is different

line three

<ctrl-D>

Now, compare the files and print the return value:

```
cmp -s file1 file2
echo $?
```

This will print 1 (one) since the files are not the same. To process a second command based on the result returned by the first, type:

```
cmp -s file1 file2 || cat file2
```

The characters `||` signify that the following command `cat` should be executed if the `cmp` command returns a non-zero value, which it will for this example.

The two characters `&&` will execute the command that follows it only if the preceding command returns a zero value. Create a third file with the command

```
cp file1 file3
```

and type the command

```
cmp -s file1 file3 && rm file3
```

It will remove `file3` if the compare command `cmp` indicates no differences. Since the compare command is preceded with a copy command `cp`, the files `file1` and `file3` have no differences, and `file3` will be removed.

### Control flow

Since the shell is a programming language, it provides conditional and looping constructs. These are `for`, `if`, `while`, and `case`. Also, a subshell can be executed within '( ' and ')'.  
The `for` construct can be used to process a set of commands, once for each element in a list of items.

To illustrate the use of `for`, type the following commands to COHERENT:

```
for i in a b c
do echo $i
done
```

The items `a`, `b`, and `c` form a list of values to be taken on by `i`. The command `echo` will be executed with `i` taking on each value in the list in turn. The result of these commands is

```
a
b
c
```

Notice that after you type the line containing `for`, COHERENT will prompt with a different character `>` (on most COHERENT systems). The shell does this to remind you that there is more information to be typed in. After you type the line containing `done`, the prompt will again become `$`.

The `for` command is usually used within a script file. Also, the list of values for the index variable can be left off, in which case the list is presumed to be the parameters to the script. To illustrate, type

```
ed
n
for i
do echo $i
echo '---'
done
w script.for
q
chmod +x script.for
```

The

```
for i
```

command is equivalent to

```
for i in $*
```

where `$*` means "all positional parameters". Notice that there are two commands to be repeated for each value of `i`. Call this script by

```
script.for 1 2 3 4 test
and the result will be
```

- 1
- 
- 2
- 
- 3
- 
- 4
- 
- lest
- 

Conditional command processing is provided with the `if` shell command. The `if` will test the result of a command and conditionally execute other commands based upon that result. You can rewrite the examples above that use `&&` and `||`. Instead of

```
cmp -s fl1e1 fl1e2 && cat fl1e2
you can use
```

```
if cmp -s fl1e1 fl1e2
then cat fl1e2
fi
```

for the same result. This means that

```
cat fl1e2
```

is executed if the `cmp` command returns a zero or true value.

To get the same result as given by the previously illustrated

```
cmp -s fl1e1 fl1e3 && rm fl1e3
```

with the `if` statement, you will need to use the `else`:

```
if cmp -s fl1e1 fl1e3
then
else rm fl1e3
fi
```

The commands between `else` and `fi` will be executed if the result of the command following the `if` is false or non-zero. Note that there is no command following `then`.

Another part of the `if` statement will allow you to test several conditions with one `if` statement and act on the one that is true. In general terms,

```
if command1
then action1
elif command2
then action2
elif command3
then action3
else action4
```

The items labeled *command* and *action* are both commands or lists of commands.

First, `command1` is executed. If the result is true, `action1` is performed.

If the result from `command1` is not true, then `command2` is executed. If its result is true, then `action2` is performed. This process continues so long as none of the commands return a true result. If none of the command results are true, the action following the `else` is executed.

To illustrate, create a shell script that will list on your terminal only one of the three file name arguments. Use the command

```
test -f name
```

which will return a value of true if `name` is an existing non-directory file.



```

ed
^
if test -f $1
then cat $1
elif test -f $2
then cat $2
elif test -f $3
then cat $3
else echo 'None are files'
fi
^
w cat.1
q
chmod +x cat.1

```

Another looping or repetitive shell statement is the **while** statement. The commands

```

while command1
do command2
done

```

will first perform *command1*. If its result is true, *command2* is executed, and *command1* is again executed. This process continues until the result from *command1* is no longer true.

The case statement resembles the if statement in that it offers a multiple choice. To illustrate, create a script that gives a choice of listing your directory in different ways:

```

ed
^
case $1 in
1) ls -l;;
2) ls;;
3) ls;;
*) echo unknown parameter $1;;
esac
^
w dir
q
chmod +x dir

```

The words **case** and **esac** bracket the entire case statement. The effect of the command

```

dir 2
ls

```

is equivalent to

Each choice within the case statement is indicated by a string followed by ):

```
2)
```

indicates the choice for \$1 having the value 2.

The strings selecting the choices may be patterns. The '\*' choice signifies that a match should be made on any string. Notice that this resembles the use of \* to substitute any file name. An expression of the form

```
1-91)
```

in a case statement will match any digit from 1 through 9. A list of alternatives may be presented by separating the choices with a vertical bar:

```
a|b|c) command
```

Notice that each command or command list in the case choice must be terminated by the double character ::.



Summary

The shell is a command programming language which handles simple commands as well as complex commands that can iterate as well as make decisions. Three kinds of substitution are provided to increase the power of your commands.

For more information about the shell, see *sh Shell Command Language Tutorial*. For more information about commands, see the *COHERENT Command Manual*.

8. Creating and using programs

The COHERENT system provides a host of high-level languages. To assist in debugging programs, symbolic debuggers are provided for many of these languages.

The languages provided with COHERENT are:

- C
- assembler

Pascal will be provided in the near future.

C is a high-level language which has replaced assembly in environments where it is available. Programming in C gives a dramatic improvement in programmer productivity, with little loss in execution speed relative to assembly language. The COHERENT system has both native C compilers and cross compilers. Compilers are available for Z8000, PDP-11, 8088, and 8086. Other versions will be available soon.

Pascal is a high-level language, featuring strong type checking, data record handling, and well-designed control structures. While similar to C, Pascal is oriented to applications programming rather than systems programming.

as gives you the assembler for the host machine. Assembly language is used for those few programs that require a special hardware access beyond what C can give. Because of the power and flexibility of C, assembly language is now effectively dead except for certain routines deep within the system. Assemblers for other computer architectures are also available with the COHERENT system. Such assemblers are called cross assemblers.

Each of the compilers reads the program source from a file. The resulting compiled program is placed in an object file. To run a program, you simply type the name of an object file as if it were a command. In fact, most COHERENT commands that you will enter are actually object programs.

### Basic steps in COHERENT programming

The steps that are necessary to generate a program are:

- 1) Edit the program source file
- 2) Compile the source program, correcting any errors
- 3) Test and debug the program
- 4) Run the program

If you have compilation errors in step 2, or program errors in step 3 or 4, you will return to step 1.

Use `ed` to build and change the source program, the `cc` command to compile the source program and produce an object program, and `ld` to help debug the program. Although the C compiler provides a macro facility, other languages do not. Therefore, if the source program uses macros, you will use `md` to expand the macros.

This section will cover each of these steps and provide some example programs.

#### `ed` — Creating the program source

Details on the use of `ed` are covered in the *ed Interactive Editor Tutorial* in detail. This section will presume basic knowledge of `ed` commands and principles of operation.

For the first program, try a simple program that prints a short message on your terminal. To build the program, enter:

```
ed
n
msh ()
{
    printf ("COHERENT will rule the world\n");
}
w small.c
q
```

With the first line, you call the editor `ed`. You add lines to the (initially empty) file using the `n` command, and signal the end of these lines with a line containing only a period or dot. The file is then written to file `small.c` with the `w` command. The `q` command exits from `ed` and returns to COHERENT.

The program itself begins with the special word `main` which defines a function and must appear in every C program. The parentheses, here with nothing between them, are used to enclose any parameters that are passed to the function. They are required even if there are no parameters. The body of the program appears between the braces `{` and `}`.

The function `printf` is a standard part of the library of C programs. It prints formatted information on the terminal. In this case it will produce the string enclosed in the double quotes. The special character string

`\n`

means "newline". Two lines of output to the terminal can be produced by

```
"1\n1\n2\n"
put as
line 1
line 2
```

Many other formatting commands are available but will not be covered here.

#### `cc` — Compiling the program

The `cc` is used to compile C programs. This command executes all the parts of the C compiler and the associated linker `ld`. The linker combines pieces of programs and includes necessary elements from the library, such as `printf`. The linker is occasionally called from the command line, but only for more complex problems than you are trying here. To compile our test program, type the command

```
cc small.c
```

If there are any errors detected, the compiler will print the message on the terminal along with the line number containing the error. You can use this line number in `ed` to find and correct the error. The command as shown will produce a program with the name `lout`. An alternative form of the compilation command

```
cc small.c -o small
```

uses the `-o` option to name the output file `small`. The program can now be used by simply typing

```
small
```

Another option `-c` tells `cc` to only compile the program and not load it.

#### `m4` — Macro processing

To extend the capabilities of all languages, the COHERENT system provides a macro processor `m4`.

Program source for all languages is made up of character strings. Macro processors perform string replacement, whereby a string in the input file may be replaced by another string. `m4` provides parameter substitution, as well as testing values of currently available strings, and conditional processing. `m4` is unique in that you can rearrange large sections of the input text by using the macros.

#### Programming simple input and output

The first example of a COHERENT program simply printed a message on the terminal. Next, write a program to copy characters from input to output.

Using `ed` to create a source file named `copy.c`, enter the following program:

```
#include <stdio.h>
/* copy file from std input to std output */
main ()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar (c);
}
```

Then, compile it with the command

```
cc copy.c -o copy
```

The `include` statement gives the program access to standard input/output definitions.

The functions `getchar` and `putchar` work with the standard input and standard output files respectively. Normally, these standard files are assigned to the terminal. Thus, if you type the command

```
copy
```

the program will read characters from the terminal and write them back on the terminal just as you typed them. The program will continue to read from your terminal until you type `<ctrl-D>`. Try this.

You will notice that this program seems to reply to you a line at a time. This is because the COHERENT system holds the terminal input until you type a `<RETURN>` and then passes the characters on to the program. This is done to enable you to correct the line using `<ERASE>` and `<KILL>` characters before the program sees it.

Even though it is written to copy standard input to standard output, `copy` can operate on disk files as well. To do so, you will use one or both of I/O redirection characters: `>` and `<`.

For example, you can copy the program source onto the terminal by saying:

```
copy <copy.c
```

which says "run program `copy`, taking standard input from file `copy.c` and put standard output on the terminal". To make a spare copy of the source program, we can say

```
copy <copy.c >copy.r.spare
```

To illustrate other C language statements, add some processing to this program and give it a new name. Use the `copy` command to make a new file `trans.c` of the source:

```
copy <copy.c >trans.c
```

The new program is called `trans`, short for `translate`.

Next, use `ed` to add some if statements before the call to `putchar`. The complete `trans` program will then look like this:

```

#include <stdio.h>
#include <ctype.h>
/*
 * Translate input to lower case,
 * removing punctuation
 */
main ()
{
    int c;
    while
        ((c = getchhr ()) != EOF) {
        if ( !isalpha (c) ) {
            if ( !isupper (c) )
                c = tolower (c);
        }
        else if (c != '\n')
            c = ' ';
        putchar (c);
    }
}

```

The additional `if` statements serve to translate all upper case characters to the corresponding lower case characters, and translates non-alphabetic graphic characters to spaces. Such a program is useful in constructing a dictionary of words from a file containing a document.

Compile this program with the command

```
cc trans.c -o trans
```

and run it with the command

```
trans
```

It will expect input from the terminal and write the translated output back to the terminal. Don't forget to end the input with `<ctrl-d>`. For example, if you type in

```
ABcdE12#1?gh1
<ctrl-d>
```

trans will type back:

```
abcde1 gh1
```

Notice that `trans` will also change the non-graphic tab character to space.

Many programs in COHERENT will expect input from files in addition to the standard input. `ed` is an example of this. This mode of operation is easily implemented by drawing on the resources of the standard I/O library, but is beyond the scope of this introduction. Many powerful and interesting programs can be written using these two standard files.

#### make — Building larger programs

All the examples of programs thus far have been self-contained. As programs grow larger, it is usual to divide the source program up into smaller files. This can simplify editing, speed compilation, increase modularity, and enable the sharing of common functions among several different programs.

Thus, in developing the larger program, you will have several source files in your directory, possibly an *include* file or two, and the object file that results from compilation. These will be used to build the loaded program file that runs when you type its name.

To change or fix the program, you will need to edit the source programs or include files in question with `ed`, recompile the required source, and relink all the modules.

But, with a change that affects several modules, it can be tricky to remember exactly which modules need recompilation, and it can be time-consuming to recompile all modules, just to be safe.

COHERENT provides a command `make` that solves this problem. `make` examines the time a file was last modified, and the time of modification of files that it depends upon, and performs the necessary compilation or other processing. (COHERENT file system directories contain the time that each file was created or modified.)

For example, if an object file `module.o` is the result of compiling source program `module.c`, then if the `c` file was changed or created after the current version of `module.o`, `make` will force a recompilation.

Further, if `module1.c` has an include file `incl.h`, and that file is changed after compilation of `module1.c`, then `make` will force a recompilation of the source, even though the object is younger than the source.

To fill out the example, assume that you are building a program named `mod`. The program is built by the `cc` command out of three files `module1.o`, `module2.o` and `module3.o` with the command

```
cc -o mod module1.o module2.o module3.o
```

and all will be compiled by commands of the form

```
cc -c module1.c
```

which produces a file named `module1.o`. Further, assume that the following files have the indicated include files:

```
module1      incl.h  incl.h
module2      incl.h  incl.h
module3      incl.h  incl.h
```

To communicate these relationships or dependencies, create a file named `makefile` with `ed`, with the following contents:

```
mod:   module1.o module2.o module3.o
      cc -o mod module1.o module2.o module3.o
module1.o:  incl.h  incl.h module1.c
      cc -c module1.c
module2.o:  incl.h  incl.h
      cc -c module2.c
module3.o:  incl.h  incl.h
      cc -c module3.c
```

There are four entries in this file, each entry consisting of two lines.

The first line of each entry begins in the first position of the line (that is, no preceding tabs or spaces) and names the file and its constituent parts.

The first entry on the first line is the name of the file that is being built and is followed by a colon. The remaining names list the files that it depends upon.

Thus, referring to the first line from the first entry above, `mod` is said to depend upon `module1.o`, `module2.o` and `module3.o`. This

means that if any one of those files changes, `mod` must be regenerated.

The second line of each entry tells how to regenerate the file. This line is a COHERENT command and should be preceded by a tab character. From then on it should be typed exactly as it would be typed by hand.

Thus, if the file `module2.c` were changed in your directory, and you issued the command

```
make
```

`module2.c` would be recompiled, changing the file `module2.o`, which would automatically rebuild `mod`.

Similarly, if you change any of the three include files, then issue the `make` command, at least two of the source files will be recompiled, again causing `mod` to be rebuilt.

`make` determines whether or not to regenerate files by comparing the date and time of the files involved. If one of the files were missing, such as would be the case if you issued the command:

```
rm module1.o
```

then this also would cause `make` to regenerate the file.

`make` can be enormously helpful in large-scale software development efforts by correctly recognizing file relationships and regenerating dependent files where necessary.

#### db — Debugging the program

The first and most critical step to debugging programs is to not put bugs in them! The method of structured analysis, design, and programming, or the method of stepwise refinement can be very effective in substantially reducing the number of errors in a program.

One can also place `print` statements at strategic points throughout the program to display logic flow and key data values. These display statements should be designed so that they can be turned off for normal operation without removing them from the program.

On occasion, however, you may find that it is necessary to debug at the machine level. If you must, COHERENT's `db` will make it possible to do so.

`db` provides tools that make the machine program instructions visible in the most natural notation. That is, instructions are displayed in a fashion that resembles assembly language, numbers can be displayed in hexadecimal, octal, or decimal as needed, and strings of characters can be displayed in familiar graphic form. `db` can also patch a program to be run again, as well as to control the execution of a program with breakpoint and single step capabilities.

Briefly, to use `db` on a program like our sample `small` above, use the command

```
db small
```

Now you can inspect and display instructions and data in the system, control execution, and even change the instructions in the program if you are bold enough.

To examine a data segment location in the program, simply type the address of the location. `db` knows about symbols in the program, so if you want to examine the location corresponding to `main`, type

```
main
```

and `db` will type out the value in hex or octal (depending upon which is appropriate for your machine).

You can expand the display command to print many locations at one time, and choose the format of printout. To print five locations interpreted as instructions, type

```
main,521
```

where the format character `l` follows the question mark indicating format, and `5` is the count of locations to be printed.

Formats other than `l` that `db` understands include

|   |   |
|---|---|
| . | current address   |
| + | next address  |
| - | previous address  |
| b | print byte in octal   |
| c | print character; non-graphic characters to be printed in octal  |
| d | decimal integer   |
| f | floating point  |
| i | disassembled machine instruction  |
| o | octal integer   |
| O | long octal integer  |
| p | show a symbolic address   |
| s | string of characters terminated by null character (C buldts strings enclosed in double quotes this way) |
| x | hexadecimal integer   |
| X | long hexadecimal integer  |

For a complete list of formats, and other details about `db`, see the information provided by the command

```
man db
```

Each format may be made up of several of these. The display address will be incremented by the size of the displayed item.

Also available in `db` are commands. To print out the value of a symbolic address, such as that of `main`, issue the command

```
main:=
```

Errors detected by `db` are signaled by a `?`. To get more extensive description of the most recently issued error message, type

```
?:?
```

To control execution with `db`, you can set breakpoints or single step through a program, or begin execution at a specified address using the appropriate commands. A breakpoint is set by specifying a desired *halt* address, followed by `:b` thus:

```
main+4:b
```

To begin execution of your program under debug, use the `e` command:

```
addr:e
```

and if you leave off `addr`, execution will begin at the entry point of the program. If the program needs parameters, type them immediately following the `e` with no intervening space. Now, begin execution with the

```
:e
```

command. When execution reaches the instruction at `main+4`, `db` will print the address of the breakpoint and the disassembled instruction.

To single step through a program from the breakpoint on, use the command

```
:s
```

which will execute one instruction and stop, or

```
,5:s
```

to execute 5 instructions and stop. `db` will print the instruction to be executed next. The alternative form of the command `sc` behaves in the same way but will treat subroutine calls as one instruction. That is, if the next instruction is a subroutine call, a

```
:sc
```

command will stop after the subroutine returns, rather than on the first instruction of the subroutine.

To continue after a breakpoint, do

```
:c
```

You can also set a breakpoint at the return of the current routine by:

```
:br
```

To delete breakpoints, issue

```
addr:d
```

To exit debug, type

```
:q
```

### Summary

Writing and testing programs is easy under the COHERENT system. You can write a program to copy files in just a few lines. COHERENT tools help you write large programs as well.

### 9. A sample problem solved with COHERENT

This section outlines a representative information processing problem and demonstrates a simple solution for it implemented with the COHERENT system.

#### Build a dictionary

Many word processing systems used today will help check your spelling. Some of them do it by consulting an internal dictionary. How might you build such a dictionary?

A very simple method of building a dictionary from the ground up with COHERENT tools will be illustrated here. This exercise will emphasize ease of construction.

The format of the dictionary is to be one word per line, all letters lower case, with no punctuation characters or spaces to be included.

Of course, the input document can be expected to have capital letters, many punctuation marks, many words on each line, and it will certainly not be in anything resembling alphabetical order!

Thus, our problem is to transform the raw input into a dictionary.



The next step is to modify the program thus shown to solve the

```

#include <stdio.h>
#include <ctype.h>
/*
 * Translate input to lower case,
 * removing punctuation
 */
main ()
{
    int c;
    while ((c = getchar ()) != EOF) {
        if (isalpha (c)) {
            if (isupper (c))
                c = tolower (c);
        }
        else if (c != '\n')
            c = ' ';
        putchar (c);
    }
}

```

This program transforms upper case letters to lower case, and all punctuation and some graphic characters to spaces. Only end of line  $\backslash n$  of the non-graphic characters remains untranslated. `trans` takes its input from the standard input, and places the output upon the standard output.

Now, we are faced with the problem of many words per line. Another small C program word entered into file `word.c` will solve this problem for us:

```

/*
 * Copy input to output with
 * only one word per line
 */
main ()
{
    int c;
    c = getchar ();
    while (c != EOF) {
        if (c > ' ') {
            /* output graphic character */
            do
                putchar (c);
            while (((c = getchar ()) > ' ') &&
                (c != EOF));
            putchar ('\n');
        }
        else
            while (((c = getchar ()) <= ' ') &&
                (c != EOF));
    }
}

```

Note that strings of spaces, newline, and control characters are transformed to newlines. Thus, if a pair of words on the input line are separated by three spaces, the output will have one newline character between them.

Compile word with

`cc word.c -o word`

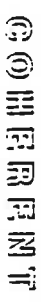
Test it with the input:

```

word
this is a test of word.
<ctrl-d>

```

The result will be



```
this
is
a
test
of
word.
```

Use a pipe to feed the standard output of `trns` to `word`:

```
trns <raw.doc | word
```

This command will list on the terminal one word per line, entirely in lower case and with punctuation removed.

Now, the result should be sorted in ascending order. The command to do so is simply:

```
sort
trns <f1 | word | sort
```

The full command will now read:

Only one more item remains to be solved. Dictionaries should contain only unique entries. The output produced so far will contain each `word` in the raw document, which means that there will be many instances of the word "the".

To perform this final bit of processing, the COHERENT program `awk` will be used to detect and eliminate duplicate lines. The *awk User's Manual* describes `awk` in detail.

`awk` is a very useful program for pattern scanning and processing. We will use only a small subset of the powerful features in `awk` for this example.

`awk` commands have two parts. The first part is the matching criterion called the *pattern*. Each input line is checked to see if it matches a pattern in any command in the `awk` program. If there is a match, the second part of the command, the *action*, is performed. The `awk` program that you will use to eliminate duplicates is:

```
$0 != prev {print; prev = $0}
```

Use `ed` to put this program into the file `u.awk`.

Each input line to `awk` is presumed to be divided into *fields*. A field is part of a line. Fields are separated by a *field separator* character, normally a space or tab. The lines in this example have only one field.

Fields are referred to by their position in the input line, preceeded with a \$ symbol. The special field `$0` signifies the entire line.

This program uses a variable `prev` that holds the value of the previous line. Each incoming line is tested for equality with the previous line by the pattern part of the statement. The command `print` outputs the new line only if it is different. Once the line is printed, the variable `prev` is set to the line just output.

The COHERENT command to call `awk` for the dictionary example is

```
awk -f u.awk >dict.s
```

The `-f` option says use the following name on the command line `u.awk` as the file name of the `awk` program. `awk` reads from the standard input.

To test this command, use

```
awk -f u.awk
```

You can type in lines and see the results on the terminal. By doing so, you can test the `awk` program.

Now all the pieces that are a solution to the problem are available. Putting them all together in one pipe command, you have a command

```
trns <raw.doc | word | sort | awk -f u.awk >dict.s
```

that will transform the raw document to a sorted dictionary. You can feed a large text file to this command to begin building your dictionary.

#### Maintaining the dictionary

Before using the dictionary, you should list it and check for extra words that you really do not want there. If the input document contains an example program, the resulting dictionary will contain program variables. You should delete any of these and other unwanted words in the dictionary.

To delete or add a few new words to the dictionary, you can use `ed` or `sed`.

#### Using the dictionary

You can use the dictionary to check the spelling of words in a new document. Create a shell file named `dict.sh`:

```
ed
n
l rns <f1.doc | word | sort | awk -F u.awk >f1.n
.
w dict.sh
q
chmod +x dict.sh
```

And process your new document with the command:

```
dict.sh new
```

This will build a file named `new.n` of unique words found in the file `new.doc`.

Now, you can use the dictionary to verify words in later documents.

First, create a shell file named `checksp`:

```
ed
n
comm -13 dict.s $1.n
.
w checksp
q
chmod +x checksp
```

This command will check a file of words, such as `new.n`, to see if there are any words that are not in your master dictionary file `dict.s`. Now use the program `comm` to give you a list of words in `new.doc` that were not in the dictionary. Type

```
checksp new
```

and any words from your document `new.doc` that were not found in the dictionary will be listed.

#### Summary of dictionary problem

This section has outlined how to build, maintain, and use a dictionary list of English words with existing COHRENT programs, and two simple, user-written C programs. The use of pipes, filters, and I/O redirection has been illustrated.

This method has been presented mainly for the purposes of illustration. It is not necessarily the best, but it is very easily implemented.

Index

\$: 81  
 %: 69, 72, 74, 76, 81, 105  
 &: 80  
 &&: 80  
 &: 62  
 #: 62, 67  
 (: 80  
 ): 85  
 \*): 85  
 \*/: 55  
 \*: 48, 55, 66-67, 85  
 + +: 55, 56  
 +: 55  
 - -: 55, 56  
 -: 55  
 . (dot): 27-28  
 .. (dot dot): 27-28, 77  
 .profile: 65, 76  
 / (division): 55  
 /=: 55  
 /: 55, 68  
 /bin: 40, 77  
 68000: 3  
 R086: 3, 87  
 R088: 3, 87  
 : (colon): 77  
 :: (double semicolon): 85  
 < ERASE>: 8, 51, 91  
 < INTERRUPT>: 8  
 < KILL>: 8, 51, 91  
 < RETURN>: 7, 9, 13, 45, 91  
 <: 17, 36, 91  
 <ctrl-D>: 9, 13, 30, 36, 43-44, 91, 92  
 > >: 37  
 >: 17, 35-36, 76, 81, 91  
 ?: 67-68, 97  
  
 @ key: 8  
 E: 68  
 V: 61  
 Vn: 89, 102  
 E: 68  
 O: 12  
 ? 75  
  
 access permissions: 41  
 arguments: 11  
 array  
 associative memory: 19  
 as: 87  
 assembly language: 20, 87  
 awk: 19, 104-105  
  
 background process: 63  
 bc: 21, 54, 55, 57  
 auto statement: 58  
 comment: 55  
 define statement: 58  
 for statement: 57  
 formula: 57  
 functions: 58  
 if statement: 57  
 name length: 57  
 names: 55  
 operator, post-  
 incrementing: 56  
 operator, pre-  
 incrementing: 56  
 operator: 55  
 program in file: 59  
 return statement: 58  
 variable: 55  
 while statement: 57  
 block, disk: 33

- brace: 73, 89
- breakpoints: 97
- C: 2, 3, 20, 57, 87-89, 102
  - program linker: 89
- cat: 21, 59
- calendar: 59
- caret: 56
- case sensitivity
  - in commands: 37
  - in file names: 23
  - in shell variable: 71
- case: 80, 84-85
  - cat: 4-5, 12, 17, 25, 29, 35-36, 38
  - cc: 88-90, 94
  - cd: 26-27, 76
  - chmod: 31-32, 64
  - choices
    - in case statements: 85
  - cmp: 19, 78-80, 82
  - cmd: 8
  - comm: 19, 106
  - commands: 11
    - COHERENT: 61
    - background: 62
    - case sensitivity: 37
    - concurrent execution: 62
    - conditional: 80, 82
    - first part: 11
    - in files: 63
    - name: 12
    - parameters: 46
    - value: 78
  - communication
    - electronic: 21
  - compiler
    - C: 89
  - cont: 8
  - control key: 8
  - conversion: 60
  - Conway, John: 3
  - cp: 25, 29
  - creating
    - files: 47
  - cross
    - assembler: 87
  - CR7: 7
  - crypt: 21, 60
  - ctrl key: 8
  - current directory: 26-28, 46
  - data entry: 19
  - data files: 18
  - date: 48
  - db: 88, 96-98
  - debugging: 95
  - default
    - directory: 76
    - permission: 32
    - prompt: 76
  - del key: 8
  - dependencies: 94
  - desk calculator: 54
  - device-independent I/O: 3, 17
  - devices: 36
  - df: 33
  - dictionary: 101
  - diff: 19
  - directory: 15, 23
    - current: 26-28, 46
    - home: 23-27, 65, 76
    - parent: 15, 27, 77
    - removing: 33
    - root: 25
    - tree-structured: 5, 15
    - user: 16
  - disk: 11
    - block: 33
    - file: 15
    - space: 33
  - do: 80
  - document preparation: 19
  - dollar sign character: 10
  - done: 80-81
  - dot command: 77
  - du: 33
  - echo: 46, 66
  - ed: 10-12, 20, 42, 47, 88-90
  - elif: 83
  - else: 82-83
  - encryption: 60
  - enter: 7
  - eof: 7
  - erase: 51
  - escape: 85
  - execute permission: 41
  - export: 74-75
  - failure: 78
  - false: 82
  - fi: 82
  - field: 105
  - separator: 105
  - file: 11, 15, 23, 36
    - attributes: 31
    - concatenation: 35
    - copying: 29
    - creating empty: 67
    - creating: 47
    - creation time: 42
    - creation: 26
    - data: 18
    - differences: 19
    - include: 93
    - input: 93
    - link: 33-34, 42
    - mode: 31
    - modification time: 93
    - moving: 28
    - name: 15, 23, 40, 42, 66
    - of commands: 63
    - output: 93
    - owner: 42
    - protection: 31
    - removal of: 32
    - size: 42
    - unvariable: 32
  - filter: 5, 17-18
  - for: 80-81
  - getchar: 91
  - GNIT: 48
  - grave accent: 75
  - grep: 19, 47-48
  - hard copy output: 45
  - hardware: 2
  - help: 13, 35
  - high-level language: 2-3, 20
  - high-level
    - language: 87
  - HOME: 76
  - home directory: 23-27, 65
  - I/O redirection: 16, 35
  - IF: 80, 82-84
  - include: 91
  - index variable: 81
  - input, standard: 47
  - inter-program communication: 17
  - keyboard: 7
  - keyword
    - Parameters: 72
  - kill: 51
  - l-out: 89
  - language
    - high level: 87
  - le: 11, 23, 25-27, 30, 39-40
    - options: 40
  - ld: 89
  - library
    - C: 89
    - standard I/O: 93
  - linefeed: 7
  - links: 33, 42
  - ln: 33
  - logging in: 8-9

- logging out: 13
- login: 13
- lower case
  - in commands: 37
  - in file names: 23
- lpr: 45
  - file: 12, 21, 39-40
  - l option: 32
- mf: 88, 90
- machine instructions: 96
- macro: 88
- mail: 21, 44-45
  - command example: 44
- mail
  - receiving: 44
- main: 89
- make: 93-95
- man: 33, 35, 97
- merge: 19
- message: 42
- mkdir: 26
- mode: 41
  - field: 41
  - of file: 31
  - msg: 21, 42-43
  - mv: 28-29
- native
  - assembler: 87
- newline
  - in C strings: 89
- nroff: 20
- o in write command: 43
- oo in write command: 43
- operating system: 1-2
- operator
  - pipe: 38
- options: 11-12, 40
- order
  - of matched file names: 68
- output formatting: 89
  - parameter: 11
    - assigning keyword: 73
    - command: 46
    - fewer: 70
    - keyword: 72
    - name: 12
    - null: 70
    - option: 12
    - positional: 69, 73-74, 81
    - references: 70
    - substitution: 78, 90
  - parameters: 12
  - parent directory: 15, 77
  - parentheses: 89
  - Pascal: 20, 87
  - password: 49
  - password: 9, 49
  - PATH: 76
  - pathname: 24-25, 28
    - fully specified: 24
    - partially specified: 24-25
  - patterns: 6, 47-48, 66, 68
  - PPP-11: 3, 87
  - performance: 4
  - permission
    - access: 41
    - execute: 41
    - read: 31, 41
    - standard: 32
    - write: 31, 41
  - pipe
    - operator: 38
  - proner: 56
  - pr: 46
  - print: 105
  - printf: 89
  - problem
    - sample: 101
- process: 63
  - background: 63
  - id: 62-63
- program
  - debugging: 95
  - modularity: 93
  - preparation: 20
  - source: 25
- programming
  - structured: 95
  - pronup: 10, 65, 76
  - protection: 1, 16, 31-32
  - PS1: 76
  - PS2: 76
  - ps: 63
  - putchar: 91
  - pwd: 28
- question mark: 67
- quit: 55
- read permission: 31, 41
- receiving mail: 44
- records: 18
- redirection: 35, 91
- regular expressions: 47
- removing
  - directories: 33
  - files: 32
- report writing: 19
- resource sharing: 1
- resources: 15
- rm: 32-34
- rmdir: 33
- root: 24, 25
- rub out key: 8
- sample problem: 101
- scat: 38
- script: 61, 64, 69, 76, 81, 83-84
- semicolons: 61
- sh: 35, 64
- shell: 35, 61
  - script: 63
  - sequential execution of com-  
mands: 61
  - simple commands: 61
  - special characters: 66
  - variable: 71, 74
- single quote: 67
- single sep: 97-98
- slash: 24
  - in pathname: 24
- software: 2
- sort: 18, 38
- space: 72
- special characters
  - shell: 61
- standard
  - /C/: 93
  - input: 17, 36, 47, 91
  - output: 16, 35-36, 91
  - permission: 32
- staff: 91
- structured
  - programming: 95
- stty: 8, 50
- subdirectory: 16
- subshell: 80
- substitution
  - in commands: 65
  - of parameters: 78, 90
- success: 78
- switches: 5
- system administrator: 8
- tab: 50, 93
- terminal: 7, 50
- test: 78, 83
- testing
  - strings: 78
- the-tac-toe: 1
- time: 48-49

User Reaction Report

To keep this manual and COURTESY free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company  
1430 W. Wrightwood Avenue  
Chicago, IL 60614

Name: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_

Phone: \_\_\_\_\_ Date: \_\_\_\_\_

Version and hardware used: \_\_\_\_\_

Did you find any errors in the manual? \_\_\_\_\_

Can you suggest any improvements to the manual? \_\_\_\_\_

Did you find any bugs in the software? \_\_\_\_\_

Can you suggest improvements or enhancements to the software? \_\_\_\_\_

Additional comments: (Please use other side.)

time  
elapsed command: 49  
zone: 48  
timesharing: 1  
tree-structured: 5

unlp: 19  
unlfs: 21, 53, 60  
unwritable file: 32

upper case  
in commands: 37  
in file names: 23  
translation: 92

user directory: 16  
user name: 8, 16, 39  
usr: 24

value from command: 78  
variable

shell: 71, 74  
vertical bar: 18, 85  
video display: 7

wait: 63  
wc: 5, 17-18  
while: 80, 84

who: 5, 16-18, 38-39, 47  
word processing: 19  
working directory: 26  
write permission: 31, 41  
write: 21, 43

ZRMMO: 3, 87

|: 17-18, 38, 85  
||: 80

