Introduction to Tripos

This manual refers to Issue 5, May 1986

Printed in the U.K

DISCLAIMER

THIS PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT THE DEVELOPER OR METACOMCO PLC OR ITS AFFILIATED DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, METACOMCO PLC OR ITS AFFILIATED COMPANIES DO NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK. IN NO EVENT WILL METACOMCO PLC OR ITS AFFILIATED COMPANIES BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Introduction to Tripos

Issue 5 (May 1986)

Table of Contents

Chapter 1: Simple Use of Tripos

This chapter provides a general overview of the Tripos operating system, including descriptions of terminal handling, the directory structure, and command use. At the end of the chapter, you'll find a simple example session with Tripos.

1.1 Chapter Overview

Tripos is a **multi-processing** operating system designed for 68000 computers. Although you can use it as a multi-user system, you normally run Tripos for a single user. The multi-processing facility lets many jobs take place simultaneously. You can also use the multi-processing facility to suspend one job while you run another.

Each Tripos **process** represents a particular process of the operating system - for example, the filing system. Only one process is running at a time, while other processes are either waiting for something to happen or have been interrupted and are waiting to be resumed. Each process has a **priority** associated with it, and the process with the highest priority that is free to run does so. Processes of lower priority run only when those of higher priority are waiting for some reason - for example, waiting for information to arrive from disk.

The standard Tripos system uses a number of processes that are not available to you, for example, the process that handles the serial line. These processes are known as private processes. Other private processes handle the terminal and the filing system on a disk drive. If the hardware configuration contains more than one disk drive, there is a process for each drive.

Tripos provides a process that you can use, called a **Command Line Interpreter** or **CLI**. There may be several CLI processes running simultaneously, numbered from 1 onwards. The CLI processes read **commands** and then execute them. All commands and user programs will run under any CLI. To make additional CLI processes, you use the NEWCLI or RUN commands. To remove a CLI process, you use the ENDCLI command. (See Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual* for a full description of these commands.)

1.2 Terminal Handling

You can direct information that you enter at the terminal to a Command Line Interpreter (CLI), that tells Tripos to load a program, or you can direct the information to a program running under that CLI. In either case, a **terminal** (or **console**) **handler** processes input and output. This terminal handler also performs local line editing and certain other functions. You can type ahead as much as you like.

To correct mistakes, you press the RUBOUT, DEL, or BACKSPACE key. This erases the last character you typed. To rub out an entire line, hold down the CTRL key while you press X. This **control combination** is referred to from this point on in the manual as CTRL-X.

If you type anything, Tripos waits until you have finished typing before displaying any other output. Because Tripos waits for you to finish, you can type ahead without your input and output becoming inter-mixed. Tripos recognizes that you have finished a line when you press the RETURN key. You can also tell Tripos that you have finished with a line by cancelling it. To cancel a line, you can either press CTRL-X or press any of the deletion keys (BACKSPACE, DEL, RUBOUT) until all the characters on the line have been erased. Once Tripos is satisfied that you have finished, it starts to display the output that it was holding back. If you wish to stop the output so that you can read it, simply type any character (pressing the space bar is the easiest), and the output stops. To restart output, press a deletion key, CTRL-X, or RETURN. Pressing RETURN causes Tripos to try to execute the command line typed after the current program exits.

Tripos recognizes CTRL-\ as an end-of-file indicator. In certain circumstances, you use this combination to terminate an input file. (For a circumstance when you would use CTRL-\, see Section 1.3.6.)

Finally, Tripos recognizes all commands and **arguments** typed in either upper or lower case. Tripos displays a **filename** with the characters in the case used when it was created, but finds the file no matter what combination of cases you use to specify the filename.

1.3. Using the Filing System

This section describes the Tripos filing system. In particular, it explains how to name, organize, and recall your files.

A file is the smallest named object used by Tripos. The simplest identification of a file is by its filename, discussed below in Section 1.3.1, "Naming Files." However, it may be necessary to identify a file more fully. Such an identification may include the device or volume name, and/or directory name(s) as well as the filename. These will be discussed in following sections.

1.3.1 Naming Files

Tripos holds information on disks in a number of **files**, named so that you can identify and recall them. The filing system allows filenames to have up to thirty characters, where the characters may be any printing character except slash (/) and colon (:). This means that you can include space ( ), equals (=), plus (+), and double quote ("), all special characters recognized by the CLI, within a filename. However, if you use these special characters, you must enclose the entire filename with double quotes. To introduce a double quote character within a filename, you must type an asterisk (*) immediately before that character. In addition, to introduce an asterisk, you must type another asterisk. This means that a file named

```
A*B = C"
```

should be typed as follows:

```
"A**B = C*""
```

in order for the CLI to accept it.

Note: This use of the asterisk is in contrast to many other operating systems where it is used as a universal **wild card**. An asterisk by itself in Tripos, represents the keyboard and screen. For example,

```
COPY filename to *
```

copies the filename to the screen.

Avoid spaces before or after filenames because they may cause confusion.

1.3.2 Using Directories

The filing system also allows the use of **directories** as a way to group files together into logical units. For example, you may use two different directories to separate program source from program documentation, or to keep files belonging to one person distinct from those belonging to another.

Each file on a disk must belong to a directory. An empty disk contains one directory, called the **root directory**. If you create a file on an empty disk, then that file belongs to this root directory. However, directories may themselves contain further directories. Each directory may therefore contain files, or yet more directories, or a mixture of both. Any filename is unique only within the directory it belongs to, so that the file 'fred' in the directory 'bill' is a completely different file from the one called 'fred' in the directory 'mary'.

This filing structure means that two people sharing a disk do not have to worry about accidentally overwriting files created by someone else, as long as they always create files in their own directories.

WARNING: When you create a file with a filename that already exists, Tripos deletes the previous contents of that file. No message to that effect appears on the the screen.

You can also use this directory structure to organize information on the disk, keeping different sorts of files in different directories.

An example might help to clarify this. Consider a disk that contains two directories, called 'bill' and 'mary'. The directory 'bill' contains two files, called 'text' and 'letter'. The directory 'mary' contains a file called 'data' and two directories called 'letter' and 'invoice'. These subdirectories each contain a file called 'jun18'. Figure 1-A represents this structure as follows:

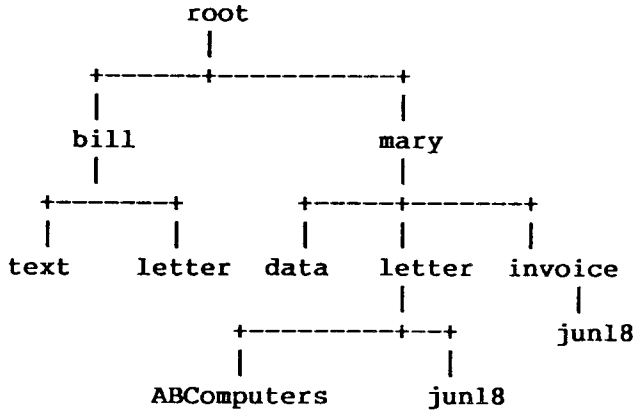


Figure 1-A: Using Directory Structure

Note: The directory 'bill' has a file called 'letter,' while the directory 'mary' contains a directory called 'letter'. However, there is no confusion here because both files are in different directories. There is no limit to the depth that you can 'nest' directories.

To specify a file fully, you must include the directory that owns it, the directory owning that directory, and so on. To specify a file, you give the names of all the directories on the **path** to the desired file. To separate each directory name from the next directory or file name, you type a following slash (/). Thus, the full specification of the data files on the disk shown in Figure 1-A above is as follows:

```
bill/text
bill/letter
mary/data
mary/letter/ABComputers
mary/letter/jun18
mary/invoice/jun18
```

1.3.3 Setting the Current Directory

A full file description can get extremely cumbersome to type, so the filing system maintains the idea of a **current directory**. The filing system searches for files in this current directory. To specify the current directory, you use the CD (Current Directory) command. If you have set 'mary' as your current directory, then the following names would be sufficient to specify the files in that directory:

```
data
letter/jun18
invoice/jun18
```

You can set any directory as the current directory. To specify any files within that directory, simply type the name of the file. To specify files within subdirectories, you need to type the names of the directories on the path from the current directory specified.

All the files on the disk are still available even though you've set up a current directory. To instruct Tripos to search through the directories from the root directory, you type a colon (:) at the beginning of the file description. Thus, when your file description has the current directory set to 'mary', you can also obtain the file 'data' by typing the description ':mary/data'. Using the current directory method simply saves typing, because all you have to do is specify the filename 'data'.

To obtain the other files on the disk, first type ':bill/text' and ':bill/letter' respectively. Another way might be to CD or type / before a filename. Slash does not mean 'root' as in some systems, but refers to the directory above the current directory. Tripos allows multiple slashes. Each slash refers to the level above. So a Unix (TM) . . / is a / in Tripos. Similarly, an MS-DOS (TM) . . \ is a / in Tripos. Thus, if the current directory is

':mary/letter', you may specify the file ':mary/invoice/jun18' as '/invoice/jun18'. To refer to the files in ':bill', you could type

```
CD :bill
```

or

```
CD //bill
```

Then you could specify any file in 'bill' with a single filename. Of course, you could always use the // feature to refer directly to a specific file. For example,

```
TYPE //bill/letter
```

displays the file without your first setting 'bill' as the current directory. To go straight to the root level, always type a colon (:) followed by a directory name. If you use slashes, you must know the exact number of levels back desired.

1.3.4 Setting the Current Device

Finally, you may have many disk drives available. Each disk device has a name, in the form DFn (for example, DF1), where the 'n' refers to the number of the device. (Currently, Tripos accepts the device names DF0 to DF3). Each individual disk is also associated with a unique name, known as a volume name (see below for more details).

In addition, the logical device SYS: is assigned to the disk you started the system up from. You can use this name in place of a disk device name (like DF0:).

The current directory is also associated with a **current drive**, the drive where you may find the directory. As you know, prefacing a file description with a colon serves to identify the root directory of the current drive. However, to give the root directory of a specific drive, you precede the colon with the drive name. Thus, you have yet another way of specifying the file 'data' in directory 'mary', that is 'DF1:mary/data'. This assumes that you have inserted the disk into drive DF1. So, to reference

a file on the drive DF0 called 'project-report' in directory 'peter',you would type 'DF0:peter/project-report', no matter which directory you had set as the current one.

Note: When you refer to a disk drive or a device, on its own or with a directory name, you should always type the colon, for example, DF1:.

Figure 1-B illustrates the structure of a file description. Figure 1-C gives some examples of valid file descriptions.

Left of the :	Right of the :	Right of a /
Device name	Directory name	Subdirectory name
-or-	*-or-*	*-or-*
Volume name	Filename	Filename

Figure 1-B: The Structure of a File Description

```
SYS:commands
DF0:bill
DF1:mary/letter
DF2:mary/letter/jun18
DOC:report/section1/figures
C:cls
```

Figure 1-C: Examples of File Descriptions

To gain access to a file on a particular disk, you can type its unique name, which is known as the disk's **volume name**, instead of the device name. For instance, if the file is on the disk 'MCC', you can specify the same file by typing the name 'MCC:peter/project-report'. You can use the volume name to refer to a disk regardless of the drive it is in. You assign a volume name to a disk when you format it (see "FORMAT" in Chapter 1,"Tripos Commands," in the *Tripos User's Reference Manual* for further details).

A device name, unlike a volume name, is not really part of the name. For example, Tripos can read a file you created on DF0: from another drive, such as DF1:, if you place the disk in that drive, assuming of course that the drives are interchangeable. That is, if you create a file called 'bill' on a disk in drive DF0:, the file is known as 'DF0:bill'. If you move the disk to drive DF1:, Tripos can still read the file, which is then 'DF1:bill'.

1.3.5 Attaching a Filenote

Although a filename can give some information about its contents, it is often necessary to look in the file itself to find out more. Tripos provides a simple solution to this problem. You can use the command called FILENOTE to attach an associated comment. You can make up a comment of up to 80 characters (you must enclose comments containing spaces in double quotes). Anything can be put in a file comment: the day of the file's creation, whether or not a bug has been fixed, the version number of a program, and anything else that may help to identify it.

You must associate a comment with a particular file - not all files have them. To attach comments, you use the FILENOTE command. If you create a new file, it will not have a comment. Even if the new file is a copy of a file that has a comment, the comment is not copied to the new file. However, any comment attached to a file which is overwritten is retained. To write a program to copy a file and its comment, you'll have to do some extra work to copy the comment.

When you rename a file, the comment associated with it doesn't change. The RENAME command only changes the name of a file. The file's contents and comment remain the same regardless of the name change. For more details, see "LIST" and "FILENOTE" in Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual*.

1.3.6 Understanding Device Names

Devices have names so that you can refer to them by name. Disk names such as DF0: are examples of **device names**. Note that you may refer to device names, like filenames, using either upper or lower case. For disks, you follow the device name by a filename because Tripos supports files on these devices. Furthermore, the filename can include directories because Tripos also supports directories.

You can also create files in memory with the device called RAM:. RAM: implements a filing system in memory that supports any of the normal filing system commands.

Note: RAM: requires the library l/ram-handler to be on the disk, and for the MOUNT command have been used to make the RAM: device available; see "MOUNT" in Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual*.

Once the device RAM: exists, you can, for instance, create a directory to copy all the commands into memory. To do this, type the following commands:

```
MOUNT ram:
MAKEDIR ram:c
COPY sys:c TO ram:c
ASSIGN C: RAM:C
```

You could then look at the output with DIR RAM:. It would include the directory 'c' (DIR lists this as c(dir)). This would make loading commands very quick but would leave little room in memory for anything else. Any files in the RAM: device are lost when you reset the machine.

Tripos also provides a number of other devices that you can use instead of a reference to a disk file. The following paragraphs describe these devices including NIL:, SER:, PAR:, and AUX:. In particular, the device NIL: is a dummy device. Tripos simply throws away output written to NIL:. While reading from NIL:, Tripos gives an immediate end-of-file. For example, the following:

```
EDIT abc TO nil:
```

allows you to use the editor to browse through a file, while Tripos throws away the edited output.

You use the device called SER: to refer to any device connected to the serial line (often a printer). Thus, you would type the following command sequence:

```
COPY xyz TO ser:
```

to instruct Tripos to send the contents of the file 'xyz' down the serial line. Note that the serial device only copies in multiples of 400 bytes at a time. Copying with SER: can therefore appear granular.

The device PAR: refers to the parallel port in the same way.

Tripos also provides the device AUX:. This device refers to the serial line, like SER:. However, unlike SER:, AUX: always treats the serial line as another interactive terminal. You usually use AUX: with the NEWCLI command; see "NEWCLI" in Chapter 1 of the *Tripos User's Reference Manual* for a full specification of this command.

There is one special name, which is * (asterisk). You use this to refer to the screen, both for input or for output. You can use the COPY command to copy from one file to another. For instance, using *, you can copy from a file to the screen, for example,

```
COPY bill/letter TO *
```

You can also copy from the screen to a file. For example,

```
COPY * TO mary/letter
```

sends anything you type on the keyboard to the file 'mary/letter'. Tripos finishes copying when it comes to the end of the file. To tell Tripos to stop copying from *, you must give the CTRL-\ combination. Note that * is NOT the universal wild card.

1.3.7 Using Directory Conventions and Logical Devices

In addition to the aforementioned physical devices, Tripos supports a variety of useful **logical devices**. Tripos uses these devices to find the files that your programs require from time to time. (So that your programs can refer to a standard device name regardless of where the file actually is.) All of these 'logical devices' may be reassigned by you to reference any directory.

The logical devices described in this section are as follows:

Name	Description	Directory
SYS:	System disk root directory	:
C:	Command library	:C
L:	Library directory	:L
S:	Sequence library	:S
DEVS:	Device for Open Device calls	:DEVS
	Temporary workspace	:T

Figure 1-D: Logical Devices

Logical device name: `SYS:`

Typical directory name: `My.Boot.Disk:`

'SYS' represents the SYStem disk root directory. When you first start up the system, Tripos assigns SYS: to the root directory name of the disk in DF0:. If, for instance, the disk in drive DF0: has the volume name My.Boot.Disk, then Tripos assigns SYS: to My.Boot.Disk:. After this assignment, any programs that refer to SYS: use that disk's root directory.

Logical device name: C:

Typical directory name: My.Boot.Disk:c

'C' represents the Commands directory. When you type a command to the CLI (DIR <cr>, for example), Tripos first searches for that command in your current directory. If the system cannot find the command in the current directory, or in any other directory in your current path, it then looks for 'C:DIR'. So that, if you have assigned 'C:' to another directory (for example, 'Boot_disk:c'), Tripos reads and executes from 'Boot_disk:c/DIR'.

Logical device name: L:

Typical directory name: My.Boot.Disk:l

'L' represents the Library directory. This directory keeps the overlays for large commands and non-resident parts of the operating system. For instance, the disk-based run-time libraries (Ram-Handler, Port-Handler, Disk-Validator, and so forth) are kept here. Tripos requires this directory to operate.

Logical device name: S:

Typical directory name: My.Boot.Disk:s

'S' represents the Sequence library. Sequence files contain command sequences that the C command searches for and uses. C first looks for the sequence (or batch) file in your current directory. If C cannot find it there, it looks in the directory that you have assigned S: to.

Logical device name: DEVS:

Typical directory name: My.Boot.Disk:DEVS

Open Device calls look here for the device if it is not already loaded in memory.

Note: In addition to the above assignable directories, many programs open files in the ':T' directory. As you recall, you find file (or directory) names predicated with a ':' in the root directory. Therefore ':T' is the directory T, within the root, on the current disk. You use this directory to store temporary files. Programs such as editors place their temporary work files, or backup copies of the last file edited, in this directory. If you run out of space on a disk, this is one of the first places you should look for files that are no longer needed.

When the system is first booted, Tripos initially assigns C: to the :C directory. This means that if you boot with a disk that you had formatted by issuing the command:

```
FORMAT DRIVE DF0: NAME "My.Boot.Disk"
```

SYS: is assigned to 'My.Boot.Disk'. The 'logical device' C: is assigned to the C directory on the same disk (that is, My.Boot.Disk:c). Likewise, the following assignments are made

```
C:          My.Boot.Disk:c
L:          My.Boot.Disk:l
S:          My.Boot.Disk:s
DEVS:       My.Boot.Disk:devs
```

If a directory is not present, the corresponding logical device is assigned to the root directory.

If you are so lucky as to have a hard disk (called HD0:) and you want to use the system files on it, you must issue the following commands to the system:

```
ASSIGN SYS:          HD0:
ASSIGN C:            HD0:C
ASSIGN L:            HD0:L
ASSIGN S:            HD0:S
ASSIGN DEVS:         HD0:DEVS
```

Please keep in mind that assignments are global to all CLI processes.

If you want your commands to load faster (and you have memory 'to burn'), type

```
MAKEDIR RAM:C
COPY SYS:C RAM:C ALL
ASSIGN C: RAM:C
```

This copies all of the normal Tripos commands to the RAM disk and reassigns the commands directory so that the system finds them there.

1.4 Using Tripos Commands

A Tripos command consists of the command-name and its arguments, if any. To execute a Tripos command, you type the command-name and its arguments after the CLI prompt.

When you type a command name, the command runs as part of the Command Line Interpreter (CLI). You can type other command names ahead, but Tripos does not execute them until the current command has finished. When a command has finished, the current CLI prompt appears. In this case, the command is running interactively.

WARNING: If you run a command interactively and it fails, Tripos continues to execute the next command you typed anyway. Therefore, it can be dangerous to type many commands ahead. For example, if you type

```
COPY a TO b
DELETE a
```

and the COPY command fails (perhaps because the disk is full), then DELETE executes and you lose your file.

The CLI prompt is initially n>, where n is the number of the CLI process. However, it can be changed to something else with the PROMPT command. (See "PROMPT" "Tripos Commands" in the *Tripos User's Reference Manual* for further details.)

1.4.1 Common Tripos Commands

This subsection describes in full the following commands, although a formal specification of each of them can be found in Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual:*

- LIST
- COPY
- TYPE
- DELETE
- WHY
- MAKEDIR
- DIR
- DATE

Even if you are only a novice user of Tripos, you'll need to give these commands at some point, so that it is important that you understand what they can do, and when and how to use them.

The examples in this subsection refer to Figure 1-A.

LIST

To find out what files are available, you can use the LIST command as follows:

```
LIST
```

This displays a list of all the files or subdirectories in your current directory. For example, if the current directory is :mary, the following is listed:

```
letter
invoice
```

'letter' in this case would be flagged as being a directory. Suppose :mary is your current directory, and you wish to list the contents of the 'letter' directory, you could, for example, specify it after the LIST command:

```
LIST letter
```

This would list the contents of the 'letter' directory on the screen.

If you attempt to LIST a file or directory which is not in your current directory, and you omit to specify the path back to the root (that is, back to the colon), an error occurs. For example, if you specify

```
LIST invoice
```

while still in :mary, the following message is displayed

```
Can't examine "invoice": object not found
```

This does not mean that 'invoice' has been lost or deleted, but simply that it cannot be found in the current directory. This can be disturbing until the filing system becomes second nature. In this case you could either reset the current directory with CD, or specify the file's path. (See also Section 1.3.3, "Setting the Current Directory", for further details.)

COPY

COPY makes an exact copy of a file or directory. The first file or directory is written to the second one using the keyword TO:

```
COPY :bill/doc TO :bill/newdoc
```

Suppose :bill is your current directory, you can alter, or even delete, 'newdoc' and still have his orginal safe in 'doc'. So long as you give the correct path, you can even send a copy of the file 'doc' to :mary by typing

```
COPY :bill/doc TO :mary/doc
```

Note that it is possible for two files with the same name to exist if they belong to separate directories. However, you must be careful if you copy a file with the same name from one directory to another as the TO file is always overwritten. For example, if you typed the following:

```
COPY :mary/doc TO :bill/doc
```

after altering the second version of 'doc' in :mary, you would lose the original. That is to say, :bill/doc would then have the same contents as :mary/doc.

To copy a directory to another directory, you could type, for example,

```
COPY letter TO invoice
```

which places a copy of the files in 'letter' in the directory 'invoice.' If there is a file in 'letter' that has the same name as a file in 'invoice' (for example, 'jun18'), then the contents of the file in 'invoice' are overwritten and lost.

To make a copy of one file and place it in another directory, you could type, for example,

```
COPY :mary.letter.jun18 TO :mary.invoice.letterJun18
```

By altering the name of the file in the second directory, you avoid the problem of overwriting the file 'invoice/jun18.'

COPY takes two possible keywords as arguments: ALL and QUIET. You only use these arguments when you use COPY with directories. For instance, you use ALL to ensure that COPY copies all the subdirectories and files on the path below the first directory name to the second. For example, if you had a directory :fred and you copied the contents of :mary to it,

```
COPY :mary TO :fred ALL
```

COPY would copy all the files and subdirectories below :mary, and any files and subdirectories in them, creating new directories as necessary. This means that ':mary/letter' and ':mary/invoice' would become :fred/letter and :fred/invoice, and all their files would also be copied over to the new directories.

Normally, when you give the COPY command to copy the contents of a directory, the name of the file being copied is displayed, followed by the word "copied" when the file is sucessfully copied:

```
COPY :bill TO :fred
text..copied
letter..copied
```

However, if the directory is large, you may not wish to have the screen filled with messages. In this case, you can use QUIET to turn off the verification. You can also use both keywords together; for example,

```
COPY :mary TO :fred ALL QUIET
```

copies the contents of the files and directories below :mary to :fred, as explained above, only without displaying the "..copied" messages.

TYPE

Once you have made a file, you can type it out (that is, display it) on the screen with the command TYPE. For example,

```
TYPE :bill/doc
```

displays the contents of the file :bill/doc on the screen. If you want your output to include line numbers, you must specify 'n' after the keyword OPT. For example,

```
TYPE :bill/doc OPT n
```

that is to say, 'type out the file with the option numbers turned on.'

If you want your file to appear as hexadecimal numbers, type h after the keyword OPT instead of n. Of course, this is more useful for files containing program code than for files containing text.

By default, TYPE types the file you specified to the current output stream (usually the screen). However, it is possible to use TYPE to 'type' one file to another:

```
TYPE :bill/doc TO :bill/newdoc
```

This is identical in effect to copying the file; 'doc' is retained and its contents is copied to the new file 'newdoc'.

You can also type a file to a device. For example,

```
TYPE :bill/doc TO PAR:
```

prints the specified file on the printer, and

```
TYPE :bill/doc TO *
```

displays the specifed file on the screen.

The speed at which a file is typed may be too fast to read if the file is longer than will fit comfortably on the screen. As explained above, you

can always suspend the output by pressing the space bar (or any other convenient character), and resume it again by pressing the RETURN key, any deletion key (DEL, RUBOUT, BACKSPACE), or CTRL-X. You can also use CTRL-S and CTRL-Q for this purpose; if CONSOLE PAGE mode is ON, the system automatically waits at the end of each page.

The only problem that might occur is if, for example, you type

```
TYPE doc
```

and the following error message appears

```
Can't open "doc"
```

This means that the file 'doc' is not in the current directory. Either the file name has been mistyped or the correct directory has not been specified with CD.

DELETE

You can use the DELETE command to get rid of unwanted files or directories. Up to ten files or directories can be deleted at one time. DELETE tries to remove each file in the order you specified. If it cannot, it gives a message and tries the next file on the list. For example, DELETE could fail because the 'object' was 'not found', or rather that the name was incorrect. A directory can be deleted if it is empty; a directory containing files may not. To delete one file, you type the name of the file you wish to delete after DELETE. For example, suppose you wish to delete the file 'doc', you would type

```
DELETE doc
```

To delete more than one file, list the filenames of the files you wish to delete after the command. For example,

```
DELETE doc newdoc letter
```

deletes the files 'doc', 'newdoc', and 'letter', provided, of course, they are in your current directory. If the files are not all in the same directory, you can still delete them by specifying their path. For example,

```
DELETE :mary/letter letter
```

deletes the file 'letter' in the directory ':mary' as well as the file 'letter' in the current directory.

If you delete all the files in a directory, you end up with an empty directory. Once a directory is empty it can be deleted. For example, suppose the directory :bill is empty and you wish to delete it, you can type

```
DELETE :bill
```

to remove the whole directory. Of course, if :bill is your current directory, you should use CD to make another directory your current directory before you try to delete it as Tripos won't allow you to delete your current directory.

WHY

It is not always convenient to consult the manual when something unexpected happens. When a command fails it gives a short error message to say something has gone wrong, although it usually does not go into any detail. There is, however, a useful command called WHY which gives further information. The message given does not go as far as the manual, but can give you a hint about what happened. For instance, if the command

```
DELETE :bill
```

fails, and it returns the message

```
Can't delete "bill"
```

you might wonder what caused the failure. Usually you can guess what has gone wrong. However, if you need more help, type WHY on a line by

itself after the failed command and a fuller message will be displayed describing what has gone wrong. For example:

```
WHY
Last command failed because object in use
```

MAKEDIR

Files may be created by ED, or EDIT; directories can only be created with MAKEDIR. You specify the name of the directory you wish to create after the MAKEDIR command. For example,

```
MAKEDIR :fred
```

creates the new directory 'fred' in the root directory.

MAKEDIR only works with one directory at a time, so you must make each directory separately. Also, for this reason, all directories on a path must exist and cannot be made at the same time. Therefore, in order to create 'ABComputers', both ':mary' and 'letter' would have to be there already. Unless you specify the path, MAKEDIR makes the new directory a subdirectory of the current directory. If you have made :mary your current directory, you can type

```
MAKEDIR letter
```

to make the directory :mary.letter. You can then either type

```
CD letters
```

and type

```
MAKEDIR ABComputers
```

or you can stay in :mary and type

```
MAKEDIR :mary/letter/ABComputers
```

The results will be the same.

DIR

The DIR command sorts all the files and subdirectories in a directory and then lists them. It can also sort and list the files in any subdirectories. If you wish, you can use DIR in interactive mode to deal with each file as it is listed (that is, examine it, delete it, and so on) or quit.

Unless you specify a directory name, DIR assumes that the directory to be listed is the current directory. Otherwise DIR lists the files in the specified directory. The order in which DIR displays the contents of a directory is as follows:

1. subdirectories (if there are any)

2. files

DIR lists the files in two columns so that you can view even a large directory at once. Of course, if you have a very large directory with a great number of files, you will never be able to view them all at the same time. In this case it is a good idea to LIST the files to a :T file. You can then view the contents of your directory by editing this file with the screen editor.

DIR can take various options after the OPT keyword; these include the A, D, and I options. Each of these options is described below.

OPT A lists the subdirectories below the specified one, indenting each sublist. For example, if you were to use DIR OPT A at the root level of the hierarchy described in Figure 1-A, then something like this would be displayed:

```
mary(dir)              bill(dir)
     data                   text
     letter(dir)            letter
          jun18
          ABComputers
     invoice(dir)
          jun18
```

OPT D only lists the subdirectory names. For example,

```
DIR :mary OPT D
```

results in

```
letter (dir)
```

if 'letter' is the only directory below :mary.

OPT I ensures DIR executes in interactive mode. When you specify interactive mode with OPT I, DIR lists each file and directory in turn. After each name, DIR displays a question mark (?) and waits for you to respond. DIR recognizes several possible answers. Each of these is described below.

If you press RETURN, DIR moves on to the next name.

If you type the letter Q, you quit DIR and the listing stops.

If you type the letter B, DIR goes back to the previous directory level, you can then type B again to go back to the level before that, and so on until you reach the level of the initial directory. You cannot, however, use B to move back beyond the initial level; that is, you cannot list files and directories above your current directory.

If DIR lists a directory name in interactive mode, then you can type the letter E to 'enter' that directory and list all the files and subdirectories. Of course, you do not type E after a filename, only after a directory.

To delete a file, type the letters DEL after the question mark. The file will be deleted immediately. You can also use DEL to delete a directory,

always providing that the directory is empty, of course. Notice that you type DEL, and that you do not press the DEL key.

If you type the letter T, DIR displays ('types') the file on the screen. CTRL-C will stop the output, but will not return to the interactive examination.

DATE

You can use the DATE command to display or set the system date or time. That is to say, you can use DATE to check the current date and time, or you can use it to set the system date or time so that all subsequent work is associated with the correct date and time. In other words, when you LIST your current directory, you find the correct dates and times listed with the filenames.

DATE has the following form:

DATE [<date>] [<time>] [TO|VER <name>]

and the following template:

DATE "DATE,TIME,TO=VER/K"

<date> is optional and can include the day of the week (Monday, Tuesday, ...Sunday), Yesterday, Today, or Tomorrow, the day of the month (01 through 31), the first three letters of the month (Jan, Feb,...Dec), and the last two numbers in the year (that is, 86 for 1986). The earliest date possible - the dawn of time for Tripos - is January 1st, 1978 (that is, 77 refers to 2077 and not 1977).

<time> is optional and comprises the hour (00 through 23), the minutes (00 through 59), and the seconds (00 through 59).

TO and VER are equivalent. They are optional keywords that can be used to introduce a different destination for the verification of the date or time. Unless you specify otherwise, the DATE command verifies the

date and time to the screen. <name> can be a device or filename.

To obtain the date and time, type

```
DATE
```

The currently set system date and time is then displayed as follows: the day of the week (for example, Monday); the day, month, and year in the form DD-MMM-YY (for example, 28-Apr-86); the time in the form HH:MM:SS (for example, 14:05:33). It is important to note that the time is always according to 24-hour clock; that is, 02:00:00 for 2 a.m and 14:00:00 for 2 p.m.

If the date is incorrect, you can use DATE to reset it. To do this, type the correct date after DATE:

```
DATE 29-Apr-86
```

If you then give the DATE command, the following, or something like it, is displayed on the screen:

```
Tuesday 29-Apr-86 14:09:54
```

However, if you set the date to one day ahead by mistake and then, when you notice your error, set it back again, any files you altered while the date was still one day ahead are listed as being last altered 'Tomorrow,' which, of course, is perfectly logical, although seemingly impossible.

Notice that you can also type a day of the week, Yesterday, Today, or Tomorrow after DATE. For example,

```
DATE Monday
```

or

```
DATE Yesterday
```

The corresponding date is then set. If the date was the 28-Apr-86, then specifying Yesterday sets the date to Sunday 27-Apr-86. To return to Monday 28-Apr-86, type

```
DATE Tomorrow
```

The current time is kept unless a new time is specified.

When you set the time you can specify it as three sets of two digits separated by colons. In this case the first two digits are assumed to be the hour, the second two the minutes, and the last two the seconds (HH:MM:SS). If you omit the last colon and everything after it (that is, you just specify HH:MM), then the time is set to the hours and minutes specified and the seconds are assumed to be 00. If you omit everything the first colon and everything after it (that is, you just specify HH), then the time is set to the hour specified and the minutes and seconds are assumed to be 00:00.

You usually only use DATE when you insert a disk for updating. When you first insert a disk, Tripos creates a process at low priority, the restart process. This process validates the entire structure on the disk. When the restart process completes, Tripos checks to see if you have set the system date and time. If there is no time or date set, you can then use DATE as described earlier to set it; otherwise, if you leave the time and date unset, Tripos sets the system date to the date and time of the most recently created file on the inserted disk. This ensures that newer versions of files have more recent dates, even though the the actual time and date will be incorrect.

If you ask for the date and the time before the validation is complete, Tripos displays the date and time as unset. You can then either wait for the validation to complete or use DATE to enter the correct date and time. Validation should happen at once; otherwise, it should never take longer than one minute.

1.4.2 Running Commands in the Background

You can instruct Tripos to run a command, or commands, in the background. To do this, you use the RUN command. This creates a new CLI as a separate process of lower priority. In this case, Tripos executes subsequent command lines at the same time as those that have been RUN. For example, you can examine the contents of your directory at the same time as sending a copy of your text file to the printer. To do this, type

```
RUN TYPE text_file TO PAR:
LIST
```

RUN creates a new CLI and carries out your printing (by directing the output of that CLI - what would have been displayed on the screen - via the parallel port device to a printer) while you list your directory files on your original CLI's output stream (the screen).

You can ask Tripos to carry out several commands using RUN. RUN takes each command and carries it out in the given order. The line containing commands after RUN is called a command line. To terminate the command line, press RETURN. To extend your command line over several lines, type a plus sign (+) before pressing RETURN on every line except the last. For example,

```
RUN JOIN text_filel text_file2 AS text_file +
SORT text_file TO sorted_text +
TYPE sorted_text to PAR:
```

1.4.3 Executing Command Files

You can use the C command to execute command lines in a file instead of typing them in directly. The CLI reads the sequence of commands from the file until it finds an error or the end of the file. If it finds an error, Tripos does not execute subsequent commands on the RUN line or in the file used by C, unless you have used the FAILAT command. See Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual* for details on the FAILAT command. The CLI only gives prompts after executing commands that have run interactively.

1.4.4 Directing Command Input and Output

Tripos provides a way for you to redirect standard input and output. You use the > and < symbols as commands. When you type a command, Tripos usually displays the output from that command on the screen. To tell Tripos to send the output to a file, you can use the > command. To tell Tripos to accept the input to a program from a specified file rather than from the keyboard, you use the < command. The < and > commands act like traffic lights directing the flow of information. For example, to direct the output from the DATE command and write it to the file named 'text_file', you would type the following command line:

```
DATE > text_file
```

See Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual* for a full specification of the < and > symbols.

1.4.5 Interrupting Tripos

Although the BREAK key is not accepted by Tripos as a valid interrupt, you can indicate four levels of attention interrupt with CTRL-C, CTRL-D, CTRL-E, and CTRL-F. To stop the current command from whatever it was doing, press CTRL-C. The following then appears on the screen:

```
**BREAK
```

followed by your usual prompt. In some cases, such as in EDIT, pressing CTRL-C instructs the command to stop what it was doing and then to return to reading more EDIT commands. To tell the CLI to stop a command sequence initiated by the C command as soon as the current command being executed finishes, press CTRL-D. CTRL-E and CTRL-F are only used by certain commands in special cases. See the *Tripos Programmer's Reference Manual* for details.

1.4.6 Understanding Command Formats

This section explains the standard format or argument template used by most Tripos commands to specify their arguments. Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual* includes this argument template in the documentation of each of the commands. The template provides you with a great deal of flexibility in the order and form of the syntax of your commands.

The argument template specifies a list of **keywords** that you may use as synonyms, so that you type the alternatives after the keyword, and separate them with an =.

For example,

ABC,WWW,XYZ=ZZZ

specifies keywords ABC, WWW, and XYZ. The user may use keyword ZZZ as an alternative to the keyword XYZ.

These keywords specify the number and form of the arguments that the program expects. The arguments may be optional or required. If you give the arguments, you may specify them in one of two ways:

By position In this case, you provide the arguments in the same order as the keyword list indicates.

By keyword In this case, the order does not matter, and you precede each argument with the relevant keyword.

For example, if the command MYCOMMAND read from one file and wrote to another, the argument template would be

FROM,TO

You could use the command specifying the arguments by position

```
MYCOMMAND input-file output-file
```

or using the keywords:

```
MYCOMMAND FROM input-file TO output-file
MYCOMMAND TO output-file FROM input-file
```

You could also combine the positional and keyword argument specifications, for example, with the following:

```
MYCOMMAND input-file TO output-file
```

where you give the FROM argument by position, and the TO argument by keyword. Note that the following form is incorrect

```
MYCOMMAND output-file FROM input-file
```

because the command assumes that 'output-file' is the first positional argument (that is, the FROM file).

If the argument is not a single word (that is, surrounded or 'delimited' by spaces), then you must enclose it with double quotation marks ("). If the argument has the same value as one of the keywords, you must also enclose it with quotation marks. For example, the following:

```
MYCOMMAND "file name" TO "from"
```

supplies the text 'file name' as the FROM argument, and the file name 'from' as the TO argument.

The keywords in these argument lists have certain qualifiers associated with them. These qualifiers are represented by a slash (/) and a specific letter. The meanings of the qualifiers are as follows:

/A The argument is required and may not be omitted.

/K The argument must be given with the keyword and may not be used positionally.

/S The keyword is a switch (that is, a toggle) and takes no argument.

The qualifiers A and K may be combined, so that the template

DRIVE/A/K

means that you must give the argument and keyword DRIVE.

In some cases, no keywords may be given. For example, the command DELETE simply takes a number of files for Tripos to delete. In this case, you simply omit the keyword value, but the commas normally used to separate the keywords remain in the template. Thus, the template for DELETE, that can take up to ten filenames, is

,,,,,,,,

Finally, consider the command TYPE. The argument template is

FROM/A,TO,OPT/K

which means that you may give the first argument by position or by keyword, but that first argument is required. The second argument (TO) is optional, and you may omit the keyword. The OPT argument is optional, but if it is given, you must provide the keyword. So, the following are all valid forms of the TYPE command:

```
TYPE filename
TYPE FROM filename
TYPE filename TO output-file
TYPE filename output-file
TYPE TO outputfile FROM filename OPT n
TYPE filename OPT n
TYPE filename OPT n TO output-file
```

Although this manual lists all the arguments expected by the commands, you can display the argument template by simply typing the name of the command, followed by a space and a question mark (?).

If the arguments you specify do not match the template, most commands simply display the message 'Bad args' or 'Bad arguments' and stop. You must retype the command name and argument. To display on the screen

help on what arguments the command expected, you can always type a question mark (?).

1.5 An Example Session

The following is an example of a simple session using Tripos. The actual screen interaction with Tripos is indented to distinguish it from text describing the action. Note also that what the computer displays on the screen is printed in a bold typeface to distinguish it from what you type.

```
> CD
  sys:
```

If you use the CD command without any further qualification, Tripos displays the name of the current directory. (> is the usual prompt from Tripos.)

```
> CD df1:tim
```

You can also use CD to change the current directory. The command sequence listed above made the directory 'tim' on disk 'df1:' the new current directory. To gain access to files stored in this directory, you simply type the filename. You no longer need to refer to the directory structure.

```
> LIST
temp            Dir  rwed Today   08:57:16
book            Dir  rwed Today   16:39:30
doc             Dir  rwed Today   09:46:06
bench1          111  rwed Today   17:08:22
bench2          125  rwed Today   18:14:24
```

LIST requests an extended list of all the files held in the current directory (df1:tim). There are two files and three directories in this directory. The directories have the word 'Dir' in the second column; the files have their size in the second column. The letters 'r', 'w', 'e', and 'd' refer to the protection status of the particular file or directory. The letter 'r' means that you can read the file or directory, 'w' means that you write

to it, 'e' means that you can execute it, and 'd' means that you can delete it. (Currently, Tripos only uses the 'd' flag.) LIST uses the last two columns to indicate when you created a file or directory.

```
> CD doc
```

The name used here after CD has no colon (:) before it, and so Tripos makes the search for the name from the current directory rather than from the root of a filing system. The current directory is now 'df1:tim/doc'. To look at the files stored in this directory, you would use the following command:

```
> LIST
```

to display the following:

```
plan          420 rwed Today 10:06:47
chapter1     2300 rwed Today 11:45:07
```

You can use COPY to create the file 'contents' in the directory 'df1:tim/doc', and everything you type at the terminal goes into the file until you press CTRL-\. This sends a new line and end-of-file character and terminates the file.

```
> copy * to contents
The Tripos User's Manual


Chapter 1: Introduction to Tripos
CTRL-\
```

You can then examine the directory contents again to see that the file does indeed exist.

```
> LIST
contents      63 rwed Today  17:01:46
plan         420 rwed Today  10:06:47
chapter1    2300 rwed Today  11:45:07
```

To see what is in the file called 'contents', you can instruct Tripos to display the file by giving the following command:

```
> type contents
```

Tripos then displays the contents of 'contents':

```
The Tripos User's Manual


Chapter 1: Introduction to Tripos
```

Table of Contents

Chapter 2: Editing Files

This chapter introduces the two editors, ED and EDIT. A full specification of both editors can be found in the *Tripos User's Reference Manual*.

2.1 Screen Editor - ED

You can use ED to edit text files. ED is a screen editor that you can use instead of the line editor EDIT.

To edit a file with ED, you specify its name after the ED command. This file is sometimes called the FROM file. The FROM file is read into memory on entering the editor and if no file of that name exists, a new file is created. Since the FROM file is read into memory, there is a limit to the size of file that can be edited using ED. The default working space is 20000 words. Normally, ED estimates how much space it needs, and then allots it. It is possible, though, for ED to be confused by a file with a large number of short lines, in which case you must use the keyword SIZE to adjust the working space manually.

Once you are in the editor, you can use ED's editing commands. Note that certain of the local line-editing commands (for example, CTRL-X) have no effect in ED.

You must specify the terminal type before you use ED. If you forget to do so, an error occurs. To specify the terminal type, you use the VDU command. This command identifies the make of terminal to be used. There are certain terminal types that are recognized by the system, and all their keyboard characteristics are understood by the console handler. The format is as follows:

VDU <terminal type>

For example, suppose you were using a Televideo 950 terminal, you would type

```
VDU tvi
```

Known terminals makes are identified in the file DEVS:VDU. It should be possible to tailor your system and support your own terminal if it is not already supported. The method is described in the *Tripos Technical Reference Manual*.

To edit the file 'doc' with ED, type

```
ED doc
```

on the other hand, you could use the FROM keyword to identify the FROM file 'doc':

```
ED FROM doc
```

The file 'doc' can now be edited, assuming it is in existence; if not, a new file of this name is created. If 'doc' is very large, type

```
ED doc SIZE 30000
```

to ensure a work space of 30000 words.

Commands to ED fall into two distinct types. The first are known as **immediate commands**, which are commands that are executed immediately, and are specified by a single key or control combination. The second are known as **extended commands**. Extended commands are typed on the command line (last line of the screen), and are not executed until the line is finished (by pressing RETURN). Several extended commands can be listed on the command line at one time; they may be grouped together and caused to be repeated automatically. Many of the simple immediate commands have a corresponding extended version.

2.1.1 Immediate Commands

The Tripos screen editor provides single-key or combined-key commands that allow you to do the following immediately:

- control the position of the cursor
- insert text
- delete text
- scroll text
- verify text
- repeat extended commands

Each of these topics is described below.

Cursor Control

You can move the cursor with the cursor control keys. If the cursor is on the edge of the screen, ED scrolls the text horizontally to make the rest of the text visible. Vertical scrolling is done one line at a time; horizontal scrolling is done ten characters at a time. The cursor cannot be moved off the top or bottom of the file, or off the left hand margin of the text. The cursor controls are usually shown on the keyboard as -> or <-, etc. As some terminals may not have cursor control keys, you can also use the following control combinations:

Control	Action
CTRL-H	Move cursor left
CTRL-J	Move cursor down
CRTL-K	Move cursor up
CTRL-X	Move cursor right

To move the cursor to the beginning or end of the **current line** (that is, the line the cursor is pointing at), you can use the HOME key. This key first moves the cursor to the right-hand end of the line; if it is already there, it moves it left to the beginning of the line. Hence repeated pressing of the key causes the cursor to jump backwards and forwards to each edge of the line. If your VDU does not have a HOME key, you can use the control combination CTRL-] instead.

CTRL-E places the cursor at the beginning of the first line visible on the screen. If it is already there it is placed at the end of the last line visible.

The control combinations CTRL-T and CTRL-R move the cursor to the start of the next word and the space following the previous word respectively. The text is scrolled horizontally or vertically if necessary.

The TAB key can also be used to move the cursor forward. The tab positions are a multiple of the tab setting (initally 3). Where the TAB key does not exist, you can use CTRL-I to achieve the same result. (See "CONSOLE" in Chapter 1, "Tripos Commands," in the *Tripos User's Reference Manual* for further details on tab settting.)

Inserting Text

In ED, any character you type at the keyboard is inserted at the cursor position, unless you attempt to add to a line that is already 255 characters long (the maximum line length is 255 characters). Any characters to the right of the cursor are immediately shuffled up to make room for any new insertion. A line may be longer than the screen width (although less than than 255 characters). If you add to a line that is longer than the screen, ED redraws the screen and scrolls the text horizontally so that you can see the end of the line. If you move the cursor beyond the end of the text on a line (for example, by using the cursor controls), ED automatically inserts spaces between the last character and any new characters.

When you press the RETURN key, ED splits the current line at the cursor and makes a new line containing everthing that was previously to the right of the cursor; this is the new current line. Everything to the left of the cursor remains unchanged. You can, for instance, use RETURN to break a long line into two in order to make it fit across the screen. This means that you do not have to scroll the text horizontally to see the end of the line.

You can also use RETURN to insert a blank line. For instance, if you press RETURN at the end of a line, you create a new blank line which becomes the new current line. If you press RETURN at the beginning of the line, you insert a blank line above the current line, but this blank line does not become the current line.

RETURN need not be pressed at the end of the line. There is a facility to set the right hand margin, initially set at the default of 79, so that when a character is typed at the end of the line, and it reaches column 79, a new line is made. Unless the last character was a space, the half completed word is moved down onto the newly generated line. Note that this only happens if characters are inserted at the end of the line.

Delete

The deletion keys work in the same way in ED as they do in local line editing, in that they erase the character immediately to the left of the cursor and move the cursor left to the position of the erased character. The text is scrolled if necessary.

CTRL-N deletes the character at the cursor position without altering the position of the cursor. This is represented by DEL CHAR on some terminals.

When you delete a character, ED moves any characters remaining on the line to the right of the cursor leftwards to fill the space previously occupied by that character. This means that characters that were beyond the right-hand edge of the screen may become visible without scrolling.

The CTRL-O command is known as 'delete word', but its action depends on what the cursor is pointing at. If the cursor is at a space, then it deletes that space, and any others to the right, up to the beginning of the next character. If the cursor is at a character, that character and all characters up to the next space are deleted. On some terminals the INS CHAR key can be used instead.

To delete all the characters from the cursor position to the end of the current line, you can use the command CTRL-Y. This command erases the character at the cursor as well as all the characters to the right of the cursor, while keeping the cursor in the same position. This is represented by DEOL or LINE ERASE on some terminals.

The command CTRL-B removes the entire current line. The line disappears and the next text line moves up to take its place. This line then becomes the current line. Note that the cursor remains in the same position throughout. Beware of this command: you cannot retrieve the line once you've pressed CTRL-B. CTRL-B is replaced by DEL LINE on some terminals.

Scrolling

The text can be scrolled vertically line by line by moving the cursor down to the last line on the screen. Alternatively, you can scroll vertically twelve lines at a time with CTRL-U (scroll Up) and CTRL-D (scroll Down). The terms 'Up' and 'Down' can be confusing. They do not mean that you move up and down the file. If you imagine the entire file to be written on a roll of paper, then rolling up the top will reveal the text written at the below. Similarly, if the roll is unrolled, then text at the top will be uncovered. Hence, scroll (or roll) up and down.

Verification

ED makes a great effort not to display any other text on the screen besides the text in the file, so that all messages appear in the bottom message area. However, in rare circumstances, other text can appear on the screen. This does not necessarily mean that the file is in the condition shown; it is just that ED has not had time to refresh the screen. To check, you can tell ED to rewrite the text by pressing CTRL-V (Verify).

Repetition

You may wish to repeat an extended command. This is often necessary when searching for a particular section of text. A search string may seem to be unique, but is often found to occur several times in a file. As ED remembers the last command line, you can repeat it by pressing CTRL-G. For example, you can use CTRL-G to search repeatedly for the same search string until you find the correct occurrence. If the command line contains any repetition commands, ED executes the same count each time you press CTRL-G.

2.1.2 Extended Commands

To start an extended command, you press the ESCAPE key. On most terminals this key is the one labelled ESCAPE or ESC; however, on some makes of terminal, the ESCAPE character is sent as a prefix to function keys. Another key must be used instead of ESCAPE on these terminals (for example, you can use F1 on the Televideo 950). You can use this substitute key wherever ESCAPE is used within this description. (See

Chapter 4, "Installation," in the *Tripos Technical Reference Manual* for details on how to install a VDU like the Televideo 950.)

When you press the ESCAPE key, an asterisk (*) appears on the last line of the screen, which is known as the **command line**. Anything you type after pressing ESCAPE appears on this line. What you type on the command line is assumed to be an extended command. You can erase any mistakes on the command line with a deletion key (for example, RUBOUT); otherwise the line cannot be edited. You can terminate the command line by pressing either RETURN or ESCAPE. ED can only execute the command line when you press either of these keys. When you press RETURN, ED executes the command(s) on the line and returns to immediate mode; however, when you press ESCAPE, it executes the command(s) and remains in extended mode. If you press RETURN immediately after pressing ESCAPE, ED returns to immediate mode without doing anything.

Extended commands consist of one or two letters of either upper or lower case. Complex multiple commands are accepted if each part is separated by a semicolon (;). Commands can sometimes be qualified by the addition of an argument or a string. An argument can be in the form of a number (such as when you set up margins), or a string. A string is a sequence of letters, numbers, or spaces defined between terminators known as **delimiters**. A delimiter is a character (which can be anything that is not a letter, number, space, semicolon, or parenthesis), which is used to introduce and terminate the sequence. Examples of strings with valid delimiters are as follows:

```
/happy/  !23 feet! :Hello!: "1/2"
```

Note that each delimiter must match. So, although ! is used in the second example, it has no effect in the third where it is used as part of the text sequence. Also the slash (/) in the first example is a delimiter, but not in the last. It is important to chose symbols that are not going to be present in the text (for example, you cannot use slashes to delimit a string containing a file description such as :doc/tripos).

Getting Out of ED

To get out of ED and keep any alterations you may have made, press ESCAPE and then type X. After a brief pause a message appears on the command line saying that the text is being written out to file. The screen then goes blank and the prompt sign appears on the screen. You will then be out of the editor and free to list your files, and so on. What happens is that ED overwrites the FROM file with the updated version. The previous version the FROM file is kept in the file :t/ed-backup. To terminate ED immediately without writing out the buffer (that is, to throw way any alterations and keep the original version of the file), type Q after pressing ESCAPE. When you type Q a message appears on the command line asking if you want ED to ignore any changes that you may have made. This gives you a second chance before anything happens to the file. If no changes have been made, the question will not appear and ED will terminate at once leaving the file intact.

The SA (SAve) command allows you to make a 'snapshot' copy of the file without coming out of ED. SA saves the text to a named file or, in the absence of a named file, to the current file. For example:

```
SA ":doc/savedtext"
```

or

```
SA
```

This command is particularly useful in areas subject to power failure or surge. It should be noted that SA followed by Q is equivalent to the X command. Any alterations made between the SA and the Q will cause the message

```
Edits will be lost - type Y to confirm:
```

to be displayed; if no alterations have been made, you quit ED immediately and the file is saved in that state. SA is also useful because it allows you to specify a filename other than the current one. It is therefore possible to make copies at different stages and place them in different files or directories.

Undoing the Last Command

From time to time, you may give a command in error; however, should this happen, all is not lost. Instead of staring in horror at the screen, press ESCAPE followed by U (for Undo). The old version of the current line is automatically reinstated and the damage undone. What happens is that the editor keeps a copy of the current line, and then modifies it as characters are added or deleted. The modified version replaces the original when the cursor is moved off the current line. The U command simply discards the modified version and causes the old version to be used instead. This means that Undo will not work after delete line, or when the cursor has been moved off the line for any reason. If this happens it is too late to undo the damage.

Blocks

A block of text can be specified by identifying the beginning with the BS (Block Start) command, and the end with the BE (Block End) command. To do this, you move the cursor to the first line of the block, press ESCAPE, and type BS. Then move the cursor to the last line of the block, press ESCAPE, and type BE. After you have given the BS command, you can use any of the cursor commands, or execute a search before you define the end of the block with BE. Nevertheless, if you make any change to the text at this point, the block becomes undefined again.

A block can be as small as one line. For example,

```
BS;BE
```

defines the current line as the current block. A block can never start or end within a line. If you only want part of a line, you must split it before you use BS and BE. A block cannot be infinite in length. If it is too long, an error occurs. The size of block allowed is relative to the size of the file; a block containing 50 lines may be allowed by a large file, but it won't by a file of about 80 lines.

Once the block has been defined it can be moved and inserted elsewhere. To insert the block in a new position, you use the IB (Insert Block) command as follows: first move the cursor to where you wish to insert the block, then press ESCAPE and type IB. A copy of the block is

immediately inserted after the current line. To delete a block, you use the DB (Delete Block) command. You give this command in the same way as above, except that you type DB. Using the BS, BE, IB, and DB commands, you can 'cut and paste' the contents of a file into a new order.

It is also possible to move blocks between files. To write a block to a file, you use the WB (Write Block) command. To do this, press ESCAPE and type WB. For example,

```
WB /Doc/
```

writes the previously defined block to the file represented by the string delimited by the two slashes (//); that is, the block is written to the file 'Doc.' Any valid delimiters can be used; the second delimiter may be replaced with RETURN. If the file does not exist, it is created. However, if the file does exist, its contents are overwritten with a copy of the block.

The IF (Insert File) command uses the same format as WB. IF inserts a file immediately after the current line. IF and WB can be used to merge together different files.

You can also use BS and BE to mark a block in order to 'remember' a particular position in a file. You can then use the SB (Show Block) command. SB resets the screen so that the first line of an identified block appears at the top of the screen.

It is important to note that blocks refer to whole lines. A BS cannot start within a line, nor can a BE finish within one. This means that you can point the cursor at any position on a line for the block marker to include the whole line. If only part of a line is required, the line must be split before marking. Similarly, a block cannot be inserted within a line. If a block is to be inserted within a line then the line must be split before the block is marked; the block markers are lost if the line is split after setting but before insertion.

Movement

The immediate commands to scroll up and down are suitable for small amounts of movement up and down the file. However, if you want to move to the end of the file, particularly if it is a long one, scroll up (CTRL-U) is unsatisfactorily slow. The B (Bottom-of-file) command immediately writes out the last part of the file on the screen. So, the last usable line contains, where possible, the bottom line of the file. Conversely, the T (Top-of-file) command writes out the top part of the file with the first line of the file at the top of the screen. The movement happens immediately because the whole of the file is kept in memory.

There are extended commands that correspond to immediate cursor control commands. These, because they require more keys to be pressed, are not as convenient for normal cursor control. They are most useful when combined in a complex command. For example, you might have a command line that finds a line containing a string, moves to the end of that line and exchanges another string, then moves to the next line and repeats the whole operation. The commands are: N for move to the start of the Next line; P for move to the Previous line; CL for move Cursor Left; CR for move Cursor Right; CE for move Cursor to the End of the current line and CS for move the Cursor to the Start of the line.

Searching and Exchanging

To find a defined string, you can use the F (Find) command. For example,

```
F /ABC/
```

finds the string 'ABC'. The search starts immediately beyond the current cursor position and continues until the string is found, or the end of the file is reached. On locating the string, ED displays the part of the file containing it on the screen.

F only searches forwards. The BF (Backwards Find) command enables you to search backwards from the cursor. The search continues until the string is found, or the top of the file is reached. The part of the file containing the string is then written out as before. For example, the command

```
BF /ABC/
```

searches backwards through the file to find the last occurrence of the string 'ABC'. If the command is repeated, the penultimate occurrence is found, and so on until the top of the file is reached.

To exchange one string for another, you can use the E (Exchange) command. This command is followed by the string to be exchanged, with the replacement string after the last delimiter of the first string. For example, if you type

```
E /Apples/Pears/
```

the first 'Apples' to be found after the cursor is replaced with 'Pears'. After the exchange, ED moves the cursor to point after the exchanged text. Spaces are allowed as strings. For example, you can exchange a double space for a single one by typing

```
E /  / /
```

A null string, two delimiters with nothing in between, can be used for the first string. In which case the second string is inserted before the current cursor position (that is, exchange nothing for string).

You can also achieve an exchange with the EQ command. EQ (Exchange and Query) will execute the exchange or not depending on the answer you give to the query. For example, suppose you give the command

```
EQ/ise/ize/
```

then it finds 'otherwise,' moves the cursor to point at the first letter of the string (that is, the 'i'), and displays

```
Exchange?
```

on the bottom line of the screen. As 'otherwize' is clearly wrong, you answer by typing an N and no change is made. You can then press CTRL-G to find the next occurrence of the string. Suppose it finds 'recognise', and you wish to Americanize your English, you type Y and

the exchange is made. If you type anything else in answer to the query, the following is displayed:

```
Commands abandoned
```

and no exchange is made. However, ED remembers the exchange strings and so you can still repeat the EQ by pressing CTRL-G.

All these search and exchange commands recognize a difference between upper and lower case: 'ABC' is not the same as 'abc'. To search for a string that could include either upper or lower case, use the UC (Upper Case) command. This is useful if you wish to find a word that can appear either at the beginning of a sentence or in the middle - such as 'All' or 'all'. For example,

```
uc;f/all/
```

finds

```
All the King's horses
```

as well as

```
Despite the presence of all the King's horses
```

Once the UC command has been given, any search or exchange will find any combination of upper and lower case. Although it does not actually do a physical translation, ED acts as if every letter is in upper case. To return to the default, you use the LC (Lower Case) command. In other words, all lower case letters are again treated as different from upper case.

Altering Text

The E command cannot be used to insert a new line into the text. The A (After) and I (Insert) commands insert lines. To insert a blank line above the current line, you can move the cursor to the beginning of the line and press RETURN, or you can use the A command. Similarly, to insert a blank line below the current line, you can move the cursor to the end of the line and press RETURN, or you can use the I command. When you

give either of these commands the lines below the current line are shuffled down to make room for the new blank line.

You can also specify a string after either the I or A commands This string is then inserted on the new line created by the command. For example,

```
A/The Walrus and the Oysters/
```

makes a new current line containing the text 'The Walrus and the Oysters' below the old current line.

To split a line in immediate mode, you move the cursor to the correct position and press RETURN. In extended mode, you can use the S (Split) command in the same way.

To join two lines together, move the cursor to the end of the first line, press ESCAPE, and give the J (Join) command. The second line is then written at the end of the first.

The extended version of delete is the D (Delete) command. This deletes the current line in the same way as CTRL-B.

To delete a character at the cursor in the same way as CTRL-N, you use the DC (Delete Character) command.

Repeating Commands

It is possible to repeat commands in extended mode. To repeat a command a certain number of times, you must specify the number of times before the command. For example,

```
4 E /slithy/brillig/
```

exchanges the next four occurrences of 'slithy' to 'brillig'. ED verifies the screen each time it executes the command. If there are less than four occurrences, the following message is displayed:

```
Search failed
```

If you do not know how many times the string occurs and you wish to exchange all occurrences, you can use the RP (RePeat) command. For example, to exchange all occurrences of 'slithy' to 'brillig', type

```
RP E /slithy/brillig/
```

This continues to exchange the one string for the other until it comes to a stop (for example, when it comes to the end of the file) and then it displays:

```
Search failed
```

in the usual way.

You can give more than one extended command at the same time if you separate them with semicolons. You can then use CTRL-G to repeat the entire group. The whole group should then be enclosed within parentheses. For example,

```
RP (F /bandersnatch/; 3 IB)
```

inserts the currently defined block three times on finding the string 'bandersnatch'. RP continues to repeat the command until an error is found. Typing a character while ED is repeating a command causes the repetition to be abandoned with the message:

```
Commands abandoned
```

Executing a Tripos Command in ED

It is sometimes useful to be able to carry out some other command while still in ED. For instance, you might want to know if a file exists before saving a copy to it. The command DO enables you to jump out of ED and do something (for example, list a directory) and then jump back.

```
DO "LIST :doc/savedtext"
```

Once the command DO is given, the screen clears and the action defined within the delimiters carried out. Afterwards press RETURN to rewrite the screen and return to ED.

2.2 The Line Editor - EDIT

EDIT is an alternative editor to ED. Unlike ED, EDIT processes files sequentially, line by line. Although the text is not necessarily displayed on the screen, editing a file in EDIT is similar to editing a file with extended commands. The file to be edited is known as the original, or **source**, file. EDIT's editing commands may be used to view or alter the contents of the source file. EDIT copies each line that you pass, and possibly alter, in the source file to an sequential file which known as the **destination** file. You can then choose to keep the destination file, the source file, or both.

The normal method is to move down sequentially, line by line through the source file changing, or not changing, the text. However, it is possible to move back a limited number of lines. These lines can be accessed because they have not yet been sent to the destination file, but are held in the output queue. Below it explains how the size of this queue, or window, can be changed. If the size has not been made large enough, it is possible to return to the beginning of the file and make another pass through the text.

2.2.1 Entering EDIT

To enter EDIT, you can give the EDIT command followed by certain arguments and keywords. For example, to edit the file 'doc', type

```
EDIT doc
```

You can, if you wish, use the the FROM keyword to identify the source file as follows:

```
EDIT FROM doc
```

If this keyword is omitted, then the first file name is assumed to be the FROM file by its position. The FROM file is the original (source). It can be kept unaltered if you exit from the program by typing STOP.

If you wish to keep the original and to create a new file containing any alterations you have made, then you must specify a TO (destination) file. The keyword TO is optional as the file is recognized by its position. The following command lines:

```
EDIT doc doc-new
EDIT FROM doc doc-new
EDIT doc TO doc-new
```

are all acceptable. In this case, after you exit from EDIT, 'doc' remains unchanged, and 'doc-new' contains the new version of 'doc' with any alterations. If 'doc-new' does not exist, it is created; if it does, it is overwritten. If a TO file is not specified, then a temporary file is created, which is then renamed with the same name as the original FROM file when you exit from EDIT. The old version of the FROM file is retained in the tempory file :t/edit-backup.

Commands in EDIT are accepted from the keyboard unless the keyword WITH is given. This keyword introduces a file containing the EDIT commands you wish to use on the FROM file. For example,

```
EDIT doc WITH edit-prog
```

takes commands from 'edit-prog' and applies them to the text in 'doc'. When the commands in 'edit-prog' have finished, your usual prompt returns, and 'doc' is updated (the destination file overwrites the source). This feature can be extremely useful. For example, you can use the RUN command before the command line:

```
RUN EDIT doc WITH edit-prog
```

and thereby do your editing in the background while you get on with something else. You can even set up a series of EDIT commands with WITH files in a C command file and then RUN the command file in the background:

```
RUN C with-commands
```

where 'with-commands' is a file containing the following:

```
EDIT doc1 WITH edit-prog TO doc1-new
EDIT doc2 WITH edit-prog TO doc2-new
EDIT doc3 WITH edit-prog2 TO doc3-new
```

EDIT then verifies the commands to the screen (for example, it displays the global number of any global command it executes), displays a colon (:) prompt when it completes each command line, and copies the results to the TO destination file. Therefore the above example can be seen to have worked if the following appears on the screen:

```
:::
```

All verification of command execution is sent to the terminal unless another destination has been specified by the keyword VER. For example,

```
EDIT doc1 WITH edit-prog TO doc2 VER confirm
```

takes the commands from 'edit-prog', applies them to 'doc1', copies the updated file to 'doc2', and sends the verification to 'confirm.'

If verification is to be thrown away, you can use the device NIL:

```
EDIT doc1 WITH edit-prog TO doc2 VER NIL:
```

Lastly, EDIT accepts two options, which are introduced by the keyword OPT. The two options are Pn and Wn. Pn denotes the maximum number of previous lines that can be held in the virtual 'window'. Wn sets the maximum line width. The default is P40 W120.

2.2.2 Basic use of EDIT

EDIT commands are very similar, but not identical, to ED extended commands. Commands typed at the terminal appear to act in almost the same way as extended commands in ED, even multiple commands are allowed using a semicolon (;) separator. The F (Find) command takes a string in the same way in both editors. So

```
*F/text/
```

in ED has the same effect as

```
:F/text/
```

in EDIT. In both cases the command is executed on pressing RETURN.

The most obvious difference on entering EDIT for the first time after using ED is that no text appears on the screen. Instead this is displayed:

```
Tripos Editor
:
```

The prompt : appears whenever the editor is waiting for a command. The : is like the * on the ED command line; whatever you type after it is executed as a command when you press RETURN, and if you fail to give a valid command, an error occurs.

Text only appears when a command is received. Each command must work on the line 'in hand'. This is the line which is read from the source and passed on to the destination file. While the line is 'in hand', it is known as the **current line**. Commands act on this line in the same way as on the line identified by the cursor in ED.

Note: Commands in EDIT can be given in upper or lower case: f/abc/ is the same as F/abc/. The same is true of extended commands in ED.

Lines are identified in EDIT by unique line numbers, which are used to move, or refer, to a particular line. As EDIT reads each successive line in a file, it assigns it a line number. The line number is not part of the information on that line and cannot be deleted or changed. However, any

lines that you insert while in the editor will have no number assigned to them. On reentering EDIT, you will find that line numbers are reassigned sequentially, and that the lines that had no numbers are numbered in sequence.

To move to line 5, type the M (Move) command followed by a 5:

```
M5
```

This command finds line 5 and makes it the current line. If line 5 has been passed, EDIT returns there, so long as line 5 is still held in main memory. It is important to note that because some lines may not have numbers (because they have just been inserted, for example), line 5 may not necessarily be the fifth line in the file.

Certain symbols are also accepted in lieu of actual line numbers. A period (.) is used to refer to the current line. For example,

```
M.
```

moves to the current line (which is, of course, an unnecessary, and silly, command). This shorthand for the current line, however, is most often used with deletion or insertion commands.

The asterisk (*) symbol is used to refer to the last line in the file. For example,

```
M*
```

means move to the last line of the file.

The plus (+) sign is used to refer to the furthest line forward you can move to and still return to the current line using the M command. To move as far as possible forwards to the last line in the current memory, type

```
M+
```

To move as far backwards as possible to the first line in memory, type

```
M-
```

Movement can also be effected without reference to a particular line. For instance, you can tell the editor to advance to the next line with the N (Next) command. To move to the next line, type

```
N
```

and to move down two lines, type

```
N N
```

and so on. As this is tedious, you can place a repetition number BEFORE the N to denote the number of lines you wish to advance. For example,

```
4N
```

tells EDIT to perform N four times. This command can be combined with a specific move:

```
M5;4N
```

In other words, 'Move to line 5 and then move on four lines'. As lines inserted during an editing session have no associated number, the ability to move on without specifying a number is essential.

In the same way that N is used to move to the Next line, P is used to move to the Previous line. For example,

```
P
```

moves one line back. A number before the command denotes how many times you wish to repeat the command. So,

```
4P
```

tells EDIT to perform P four times (that is, 'Move back four lines'). Although N works until the end-of-file is reached, P only works on lines held in main memory.

It is also possible to specify a particular line by its context. As mentioned above, the search command F works in the same way as in ED. The command is given followed by a string. The editor then searches forward to find a line containing that string, making the located line the new current line. The search continues until the end of the file is reached. The message displayed by ED is 'Search failed' if the context is not found; EDIT displays 'SOURCE EXHAUSTED' if it cannot find it.

Like in ED, it is also possible to search backwards in EDIT with the BF (Backwards Find) command. Again, only the lines in memory can be used. The same syntax as ED is used

```
BF /Jack and Jill/
```

(that is, 'search back through the file to find a line containing Jack and Jill'). If the string is not found, then the message

```
NO MORE PREVIOUS LINES
```

is given.

Character strings have delimiters in the same way as in ED. There are a number of characters that can be used; however, they must not be letters, numbers, spaces, semicolons, or brackets. The delimiter that is chosen must not also occur within the string's text. So,

```
fetch a pail of water.
```

should not have a period (.) as a delimiter. Spaces between the command and the first delimiter are not significant, and are ignored; spaces within the string are significant, since the context must match exactly.

```
Jack fell down
```

is different from

```
Jack   fell down
```

and so no match is made.

An F with no following string repeats the previous search. The search starts at the current line, and so, if the same command is typed again, the context in the same current line is found. To progress to find new occurrences, you must advance the current line. For example,

```
N;F
```

moves onto the next line and repeats the search forwards. Similarly, for Backwards Find,

```
P;BF
```

moves back a line and repeats the search back through the file.

EDIT has further additional refinements to the simple search command. These are known as **qualifiers**. They allow the search to be limited, for instance, to the beginning or end of the line. Thus,

```
F B/Jack/
```

finds the first line beginning with the word 'Jack'. In this case, it finds the first line of the rhyme. To find a line with a specific ending, you add the E (End) qualifier

```
F E/ water./
```

This finds a line ending in ' water.'; in this case:

```
to fetch a pail of water.
```

The strings /Jack/ and / water./ are called **qualified strings**. Another qualifier restricts the search to finding a line containing precisely a string with no other characters, either before or after the string. This is the P (Precisely) qualifier. Hence,

```
F P/to fetch a pail of water./
```

locates the line

```
to fetch a pail of water.
```

whereas

```
F P/fetch a pail/
```

does not. You can also give a null string (that is, two delimiters enclosing nothing) after 'F P':

```
F P //
```

In other words, 'Find Precisely nothing', or 'find an empty line'.

Typing Text

Sometimes it is not sufficient to get the context from the single current line. In which case it is possible to type out (that is, display) several lines using the T (Type) command. For example,

```
T
```

starts at the beginning of the current line and continues to 'type' until it reaches the end of the file. You can also specify how much you wish to be 'typed'. For example,

```
T6
```

types the next six lines, and

```
TP
```

types the lines in the output queue.

Changes on the Current Line

Exchange - To exchange one character string for another type:

```
E /Jack/John/
```

This exchanges the first occurrence of the first string on the current line for the second string, giving:

```
John and Jill
```

If /Jack/ is not found the message

```
NO MATCH
```

is given.

After - This places a second string immediately after the first occurrence of the first string on the current line. For example,

```
A /John/ny/
```

results in:

```
Johnny and Jill
```

Before - This places the second string immediately before the first occurrence of the first string on the current line. An empty string (two delimiters together with no space in between), causes the second string to be placed at the beginning of the line. For example:

```
B //Mary, /
Mary, Johnny and Jill
```

All the previous commands work by reading the line left to right, and then acting on the first occurrence. To act on the last the qualifier 'L' is used. This causes the line to be read right to left. For example:

```
E L/y/ie/
Mary, Johnnie and Jill
```

Globals

The commands E (Exchange), A (After), and B (Before) only work on one line. Each occurrence of Jack can be exchanged for John throughout by setting up a **global exchange**. A global exchange command remains active until it is cancelled. Each time a new line is read, and the string matched, the exchange is made. So

```
G E/Jack/John/
```

globally exchanges Jack for John, and the lines that contained the string 'Jack' will appear as follows:

```
John and Jill
John fell down
Up John got
```

and so on.

Line Deletion and Insertion

Delete - To delete a line, you use the D (Delete) command followed by the line number of the line you wish to remove. For example,

```
D5
```

deletes line number 5. To delete the current line, type

```
D.
```

where period (.) denotes the current line. (D without a period is a synonym for 'D.'.) It is also possible to delete lines inclusively from one numbered line to another. For example,

```
D5 7
```

deletes lines 5 to 7.

All successive lines until a line containing a specific string can be deleted by the DF (Delete Find) command. For example,

```
DF /fetch a pail/
```

deletes the lines

```
Jack and Jill
went up the hill
```

but stops when it reaches the line

```
to fetch a pail of water
```

Insert - To insert a line, you use the I (Insert) command followed by a line number or accepted symbol. For example,

```
I.
```

inserts anything you type on the keyboard before the current line. Until the insertion is terminated, all characters, including those used as commands, are treated as text to be inserted. To terminate the insertion, type the letter 'Z' on a line by itself. For example,

```
Il
Jack and Jill
went up the hill
Z
```

tells EDIT to take the two lines typed before the Z and insert them before line 1.

To insert the text of another file, you use the I (Insert) command followed by the name of the file you wish to insert in the form of a string:

```
I ":Bill/nursery-rhyme"
```

The original lines are unchanged whenever you use an insertion command. The new lines are, for the present, have no line numbers; if

you move to them, you will see that they have plus signs (+ + +) where the line number usually appears.

To remove a line and insert a replacement, you use the R (Replace) command. This command is used in exactly the same way as the insert command; it even has the same terminator. For example,

```
R.
and they all lived happily ever after.
Z
```

replaces the current line with the text typed before the Z. Notice that the text must be on a separate line from the Z.

A common mistake with both the I and R commands is to forget to type the Z; if you do forget, then nothing appears to work as EDIT thinks the commands you type are just lines of text you wish to insert.

Split and Join

Splitting a line - A line can be split before or after a string. The first part of the line being sent to output, and the second part remaining as the current line.

The SB (Split Before) command splits the line before the specified string. For example,

```
SB /Jill/
```

rewrites the line 'Jack and Jill' as:

```
Jack and
Jill
```

The SA (Split After) command splits the line after the specified string. For example,

```
SA /Jack/
```

rewrites the same line as:

```
Jack
 and Jill
```

The leading space would have been lost if the split had been

```
SA /Jack /
```

Joining lines - To join, or concatenate, a line so that the beginning of the new current line is the current line followed by the given string, ending with the next line, you use the CL (Concatenate Line) command. For example,

```
CL / /
```

puts a space at the end of the current line and then concatenates the current line and the next line to make a new, longer current line. If this command is executed when the current line is the first line of the rhyme, the result is as follows:

```
Jack and Jill went up the hill
```

A repeat counter before CL enables you to specify how many lines you wish to be concatenated: 2CL/ / causes the next two lines to be concatenated, 3CL/ / the next three lines, and so on.

2.2.3 Terminating an EDIT Session

You can can terminate EDIT in one of two ways. First, you can use the W (Windup) command; this allows you to keep all your alterations in the TO file when you leave EDIT. Second, you can use the STOP command; this stops EDIT immediately and throws away any alterations you may have made. For example,

```
W
```

W writes the source file back to the destination. If there are any oustanding globals, then they are executed as the source file is read and then written out to the destination.

STOP

immediately stops whatever command is being executed, keeps the source file intact, throws away the destination file, and terminates the EDIT session. All edits are then lost.

Table of Contents

Chapter 3: Further Use of Tripos

This chapter introduces further topics and explains in greater detail some of those raised in Chapter 1.

3.1 Tasks

Like some other modern operating systems, Tripos provides the user with the ability to perform a number of different jobs concurrently. This ability to multi-task means that single users, at one console, can act as if they have several machines at their command at the same time. These virtual machines all share the same screen; a simple escape combination enables you to switch from one to the other. Using this system, you can print out a file and compile a program, while editing another file. Instead of having one machine for several people, Tripos makes the machine act as if it is several machines for one person.

Each Tripos task represents a particular process in the operating system. (In fact, the term 'process' is a synonym for 'task' and you will find that both terms are used interchangeably in this manual.) Only one task is actually running at a time; other tasks are either waiting for something to happen, or have been interrupted and are waiting to be resumed. For each task there is an associated priority level. The task with the highest priority runs first. Lower priority tasks run only when those at a higher level are waiting for some reason - such as for data to arrive from a disk or from the keyboard.

Tripos has at least four tasks in the standard system. The first task, task 1, is the Command Line Interpreter or CLI. This accepts commands and attempts to execute them. All commands and user programs run under a CLI. To set up further CLIs, you can use the NEWCLI or RUN command. (See Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual* for a full specification of NEWCLI and RUN.)

The second task, task 2, is the debug task. You may select the debug task yourself, or you may find that the system enters it for you if something goes wrong. Once in debug, you can examine and alter the state of the computer and then continue program execution if all is well.

Task 3 and task 4 handle the terminal and the filing system on the disk. If there is more than one disk device, a separate task is devoted to each one.

Initially anything you type on the keyboard is directed to the Command Line Interpreter (CLI). Input and output is then processed by the console (terminal) handler, which performs local line editing (for example, character and line deletion).

To select another task, press CTRL-P (P for Process). This combination allow you to move from task to task. Each time you press CTRL-P you select another task. By repeatedly pressing CTRL-P you can cycle through all the available tasks.

If necessary, you can use the CONSOLE command to change the control combination to something else. That is, you can choose another character to press with the CTRL key. You should make sure, though, that the character you specify is not already used for something else. For example, it is not a good idea to exchange CTRL-P for CTRL-X; if you do, you'll lose the ability to cancel a line.

Initially input is directed to the CLI on task 1 as it handles all commands. However, you can use the CTRL-P combination to connected your terminal to other tasks. If you use NEWCLI to create a new task, then you can press CTRL-P to select it (you may have to press CTRL-P several times to do so, though). When you select another CLI, you leave the first CLI suspended. To reselect the initial CLI again, press CTRL-P as many times as is necessary until you return to the correct task. You can tell which task is which if you use the following PROMPT command:

```
PROMPT %n>
```

the prompt then includes the task number. For example,

```
8>
```

means that you are using task 8. Task 2 (debug) is easily recognized as it has its own prompt, an asterisk (*).

3.2 Commands to Tasks

This section describes the commands that let you start, stop, and restart a task. It also explains how you can examine tasks.

3.2.1 Stopping a Task

BREAK - The command BREAK (not the BREAK key), sets up specified attention flags. The flag CTRL-C is set by default; CTRL-D, CTRL-E, and CTRL-F may also be used. The result of the command BREAK is identical to selecting the relevant task and pressing the required CTRL. For example,

```
BREAK 7
```

means set the CTRL-C (default) of task 7. Alternatively, select task 7 (by pressing CTRL-P as many times as is necessary) and then press CTRL-C, the effect should be the same. To set the CTRL-D flag of task 5, type

```
BREAK 5 D
```

ENDCLI - This command terminates an interactive CLI task. That is, ENDCLI terminates the CLI currently selected by the CTRL-P combination. ENDCLI is normally used to end a CLI created by NEWCLI. If the initial CLI (task 1) is terminated, and no new one set up, then the session is automatically ended. You should use this command with care.

3.2.2 Starting and Restarting a Task

NEWCLI - This command creates a new interactive CLI task while remaining in the same currently selected task. To use the new CLI, you must press the CTRL-P combination as many times as is necessary until you obtain the correct prompt. Once you have selected a new CLI with NEWCLI and CTRL-P, you will find the current directory and prompt is the same as in your initial CLI. When a new CLI starts up it announces its task number, which you can then use to identify it.

3.2.3 Examining Tasks

STATUS - This command displays information about the tasks currently in existence. The command alone produces a list of the task numbers, with an indication as to whether they are waiting, suspended, running, interrupted, held, broken or if their work queue is empty. Information about a specific task can also be given:

```
STATUS 4
```

will show the state of task number 4. Further arguments can be given to get fuller or more specific information.

3.3 Patterns

A **pattern** is used to match certain files. It consists of several special characters which have certain meanings, and any other characters which match themselves. The special characters are as follows:

```
' ( ) ? % # |
```

The prefix ' is used to remove the special effect of any of these characters and cause them to match themselves. For instance '? matches ? and '' matches '. The action of the special characters is as follows:

?	matches any single character
%	matches the null string
#<p>	matches any number of occurrences of the pattern <p>
<p1><p2>	matches a sequence of pattern <p1> followed by <p2>
<p1>|<p2>	matches if either pattern <p1> or pattern <p2> match
()	are used to group patterns together

Thus

A#BC	matches AC, ABC, ABBC, and so on
A#(B|C)D	matches AD, ABD, ABCD, and so on
A?B	matches AAB, ABB, ACB, and so on
A#?B	matches AB, AXXB, AZXQB, and so on
'?#?'#	matches ?#, ?AB#, ??##, and so on
A(B|%)#C	matches A, ABC, ACCC, and so on

This is rather complicated but, once mastered, allows LIST and other commands to be used with extreme flexibility. For example,

```
LIST :bill PAT doc#?(x|y)
```

means "Examine the directory :bill, printing information on files which start 'doc' and which end with either 'x' or 'y'."

Many commands take patterns as arguments, some examples using familiar commands appear below. For instance, to copy all the files in the current directory which start 'test-' to the disk device 'df1:', type

```
COPY test-#? to df1:
```

Deleting files is always dangerous. If many have to be deleted it is easy to suffer from 'finger factor', or cheerfully continuing to erase files that should be kept. To avoid accidentally deleting too many files, you should consider carefully any pattern set up for deletion before you use it. An example of its use might be as follows:

```
DELETE t#?/#?(1|2)
```

which means: 'delete all files ending in 1 or 2 in directories starting with t.' This is useful to clear out the temporary directory :T. The more cautious user can always check with LIST before using DELETE.

You do not have to understand the use of patterns to use Tripos. However, they can save you a lot of typing.

3.4 Command Files

This section provides additional examples for the C command. Although C command files can be simple, they can also be highly complex. The examples in this chapter show just how sophisticated they can be.

3.4.1 Example 1

This example shows parameter substitution by keyword name and/or position.

The .KEY (or .K) statement supplies both keyword names and positions in command files. It tells C how many parameters to expect and how to interpret them. In other words, .KEY serves as a "template" for the parameter values you specify. Only one .KEY statement is allowed per command file. If present, it should be the first command line in the file.

When you enter a command line, Tripos resolves parameter substitutions for the keywords in two ways: by specification of the keyword in front of the parameter, and by the relative positions of the parameters in the line. Keyword name substitution takes precedence.

For example, the following .KEY statement:

```
.KEY flash,pan
```

tells Tripos to expect two parameter substitutions, <flash> and <pan>. (The angle brackets indicate the keyword value to be substituted at execution time.)

Suppose you enter the following command line:

```
C DEMO1 pan somename flash othername
```

The value "othername" is assigned to <flash>, and the value "somename" is assigned to <pan>.

You can omit the keyword names if the parameter substitutions are in the order given in the .KEY statement. For example, the following statement is equivalent to the preceding one:

```
C DEMO1 othername somename
```

This is because the values correspond to the keyword order specified in the .KEY statement.

You can also mix the two methods of parameter substitution. Suppose you have a .KEY statement with several parameters, as follows:

```
.KEY word1,word2,word3,word4
```

The C file processor removes parameter names from the input line to fill the meanings of any keyword values it finds. Then, with any remaining input, it fills the leftover keyword positions according to the position of the input value.

For example:

```
C DEMO2  word3 ccc word1 aaa bbb ddd
```

The processor assigns ccc to <word3>, aaa to <word1>, and has two parameters left over. Scanning from left to right in the .KEY statement, it finds that <word2> is still unassigned. Thus, <word2> gets the next input word, bbb. Finally, <word4> hasn't been assigned either, so it gets the last input word, ddd.

You can indicate special conditions for parameter substitution, as follows:

```
.KEY name1/a, name2/a, name3, name4/k
```

The "/a" indicates that a value must be supplied to fill the parameters for name1 and name2. Values for name3 and name4 are optional, though the "/k" indicates that <name4> (if supplied) must be preceded by the explicit keyword "name4." For example:

```
C DEMO3 fee fie foe name4 fum
```

If the user does not supply a required parameter (such as name1 or name2 in the preceding example), C issues an error message.

As an example of the use of the /k option, suppose you have created a command-file named COMPILE and it lets you optionally specify a file name to which a printout of the compilation is to be directed. Your .key statement might read:

```
.key compilewhat/a,printfile/k
```

If a user enters a line such as:

```
C COMPILE myfile PRINTFILE myprint
```

the command-file says the keyword PRINTFILE is optional and need not be supplied, but if used, there must be a value entered along with it. Thus the above line is correct, since myprint is specified as the target output file.

3.4.2 Example 2

This example demonstrates how you can assign default parameters and different bracket characters.

Note: the following example can be executed as a batch file.

```
.KEY word1
```

The .DEF directive establishes a default value for a keyword if the user does not specify a value on the command line. To detect an unsupplied parameter value, you can compare it to "" (two double-quotes in a row). You must perform this comparison before executing any .DEF statement in the command file.

You can assign defaults in either of two ways. The first way requires that you specify the default every time you reference a parameter, using the "$" operator.

For example, in the following statement:

```
ECHO "<word1$defword1> is the default for Word1."
```

"defword1" is the default specified for word1 and is printed when the above statement executes. The second way is to define a default once. For example, with the following assignment:

```
.DEF word1 "defword1"
```

you can execute the following statement:

```
ECHO "<word1> is the default for Word1."
```

The output of both of the above ECHO statements will be:

```
defword1 is the default for Word1.
```

Note that a second use of .DEF for a given parameter has no effect:

```
.DEF word1 "--- New default ---"
ECHO "<word1> is STILL the default for Word1."
```

(The first assignment, "defword1" will be substituted for word1 at execution time.)

Wherever C finds enclosing angle brackets, it looks within them to see if it can substitute a parameter. An unsupplied parameter with no default becomes a "null" string.

Suppose you want to use a string that contains the angle bracket characters, < and > . You can use the directives .BRA and .KET to define substitutes for the bracket characters. For example,

```
ECHO "This line does NOT print <angle> brackets."
.BRA {
.KET }
ECHO "This line DOES print <angle> brackets."
ECHO "The default for word1 is {word1}."
```

The first ECHO statement causes the processor to look for the parameter substitution for "angle," since that's the current meaning of the angle bracket characters. Since "angle" wasn't included in the .KEY statement, the processor substitutes the null string for it. Then, after the .BRA and .KET directives redefine the bracket characters, the second ECHO statement prints the characters:

```
This line DOES print <angle> brackets.
```

The third ECHO statement illustrates that the braces ({ and }) now function to enclose keywords for the purpose of parameter substitution.

3.4.3 Example 3

This example demonstrates file copy simulation showing command file structures.

The IF statement lets you peform tests and cause different actions based on the results of those tests. Among the possible tests are testing strings for equality and testing to see if a file exists. You can use an ELSE statement with an IF to specify what should be done in case the IF condition is not true. The ELSE statement, if used, is considered a part of the IF statement block. An ENDIF terminates an IF statement block.

The example programs below also use a SKIP statement. The SKIP statement lets you skip FORWARD ONLY within your command-file to a label defined by a LAB statement.

The IF...ENDIF structure is illustrated by the following short example. It is generally a good idea to test for keywords that might be omitted, or might be entered as null ("") in quotes, as shown below:

```
IF "<word1>" EQ "usage"
   SKIP USAGE
ENDIF
IF "<word2>" EQ ""
   SKIP USAGE
ENDIF
```

Enclosing your parameter substituition words in double-quotes within IF statements prevents C from reporting an error if the keyword is omitted.

If you omit the double quotes and the value is not supplied, the result can be a line that reads:

```
IF EQ "usage"
```

This produces an error, because the two operators IF and EQ are adjacent. Using double quotes around the keyword replacement indicators results in a line that reads:

```
IF "" EQ  "usage"
```

which is legal.

You can use NOT in an IF statement to reverse the meaning of the test you perform. For example:

```
IF NOT exists <from>
```

There can be nothing on the IF line other than the test condition. For example, the following is incorrect:

```
IF <something> EQ true SKIP DONE
```

The correct form of the above statement is as follows:

```
IF <something> EQ "true"
   SKIP DONE
ENDIF
```

As the example above shows, the string constant tested for need not be enclosed in double-quotesin the preceding example, either "TRUE" or TRUE is acceptable.

As shown in the sample command file below, IF statements can be nested so that commands can be executed based on multiple true statements. Note that C lets you indent to make the nesting of IF statements more readable.

The following sample command file simulates a file copying utility that illustrates certain useful structures in a command file: IF...[ELSE]...ENDIF, LAB, and SKIP.

```
.KEY from, to      ;  Assign parameter list
IF "<from>" eq "" ; Check for a FROM file
being supplied.
  SKIP usage       ; No file, show user how to use.
ENDIF
IF "<to>"   eq "" ;  Check for a TO file
being supplied.
   SKIP usage      ;  No file, show user how to use.
ENDIF

IF NOT exists <from> ; Check if FROM file
; doesn't exist
   ECHO "The FROM file you selected (<from>)"
   ECHO "could not be found."
   ECHO "Please use the DIR or LIST command"
   ECHO " and try again."
   SKIP DONE    ; Note: We can SKIP out of an IF.
ENDIF

IF exists <to>          ;  Check if TO file exists.
   IF "<o>" EQ "O"      ;  Did the user supply "O"
on the line ?
      copy from <from> to <to>
      ECHO "Replaced file named <to> with a copy of
      ECHO " file named <from>."
      ECHO "Request fulfilled."
   ELSE
      ECHO "Command overwrites an existing file "
      ECHO "ONLY if the O parameter is specified."
      ECHO "Request Denied"
      SKIP usage     ;  Explain how to use this file
   ENDIF
ELSE
      ECHO "copy from <from> to <to>."
ENDIF
SKIP DONE
```

```
LAB usage
ECHO "cp:  usage...."
ECHO "The following copy forms are supported:"
ECHO "   x cp FROM sourcefile TO destinationfile"
ECHO "   x cp FROM sourcefile destinationfile"
ECHO "   x cp sourcefile TO destinationfile"
ECHO "   x cp sourcefile destinationfile"
ECHO "   x cp TO destinationfile FROM sourcefile"
ECHO "   x cp sourcefile destinationfile O"
ECHO "   x cp FROM sourcefile TO destinationfile O"
ECHO "   x cp O FROM sourcefile TO destinationfile"
ECHO "where: x is short for Ccp is the name of"
ECHO "this command file, and *"O*" means "
ECHO "'overwrite existing file'."

LAB DONE
```

3.4.4 Example 4

This example provides a sample looping batch file.

The SKIP command allows only forward jumps. To create a loop structure within a command file, use C iteratevely. That is, use the C command within the file itself to send execution backwards to a label. The following executable example illustrates looping.

```
;This file displays five messages:
;"This prints once at the start. (p1,p2)"
;"Loop number I."
;"Loop number II."
;"Loop number III."
;"This prints once at the end. (p1,p2)"

.KEY  p1,p2,loopcnt,looplabel
FAILAT 20
IF NOT "<looplabel>" EQ ""     ; Called with label?
    SKIP <looplabel>           ; Yes, then loop.
ENDIF
```

```
ECHO "This prints once at the start. (<p1>,<p2>)"
   ; ==== Start of loop ====
LAB lst-loop
IF "<loopcnt>" EQ "III"        ; finished looping?
?
  SKIP loopend-<looplabel>     ;Yes, unwind.
ENDIF
ECHO "Loop number <loopcnt>I." ;Go 'backwards'
; in this file

C. loop.sample "<p1>" "<p2>" <loopcnt>I lst-loop

LAB loopend-<looplabel>
IF NOT "<loopcnt>" EQ ""
   SKIP EXIT
ENDIF                          ; === end of loop ====
ECHO "This prints once at the end. (<p1>,<p2>)"

LAB EXIT
```

3.5 Command Paths

The full description of a particular directory or file is called its path; it describes the complete hierarchy from the root (:) through to that directory of file. In other words, it describes the 'path' you have to go along in order to find that directory or file.

When you use a command, Tripos looks for it first in C: and, if it cannot find it there, it then looks for it in your current directory. You can, however, specify further directories for Tripos to search. In fact, the PATH command not only allows you to specify exactly which directories Tripos should search, but it also lets you specify the order in which it should search them.

You may wish to add to the default list. This can be done by typing

```
PATH ADD
```

followed by between one and ten names. For example,

```
PATH ADD :com :mary/commands :fred/commands
```

adds these three directories after the currently set ones, and in the order given (that is, :com is searched before :mary/command, which is searched before :fred/commands).

To replace the entire search list, omit the keyword ADD when you list the directory names. To clear the list, you give the command with no arguments; that is, you set the command path search list to nothing.

If you wish to find out the directories in the list, give the keyword SHOW after the command; for example,

```
PATH SHOW
> c: bill :com :mary/commands :fred/commands
```

3.6 Startup Sequence

Sequence files are held in the sequence library on the logical device S:; they contain command sequences. One sequence file that you will almost certainly use is S:startup-sequence. This file sets up all your own default commands when you startup Tripos on your computer. For example, it can set your default CONSOLE parameters and your default terminal type:

```
VDU TVI
CONSOLE PAGE ON AUTLN OFF
```

3.7 Errors

When a command fails it returns a failure code and sends a brief message to the terminal. More information about what has gone wrong can be obtained by using the command WHY.

The command FAULT accepts a failure code (also variously known as a return, fault, or error code) as an argument and returns its corresponding message. You can specify up to ten codes at a time. For example,

```
FAULT 22 221 218
```

displays the messages for faults 22, 221 and 218.

If your command fails, use FAULT or WHY. If you are still confused, you can look up the error in Appendix A, "Error Codes and Messages," in the *Tripos User's Reference Manual*. This appendix suggests what might have gone wrong and supplies a possible course of action. Not all errors are recoverable, however.

Glossary

Arguments
Additional information supplied to **commands**.

Character pointer
Pointer to the left edge of a line window in EDIT. You use it to define the part of a line that EDIT may alter.

Character string
Sequence of printable characters.

Command
An instruction you give directly to the computer.

Command Line Interpreter (CLI)
A **process** that decodes user input.

Console handler
See **terminal handler**.

Command template
The method of defining the syntax for each **command**.

Control combination
A combination of the CTRL key and a letter or symbol. The CTRL key is pressed down while the letter or symbol is typed. It appears in the documentation, for example, in the form CTRL-A.

Current cursor position
The position the cursor is currently at.

Current directory
This is either the **root directory** or the last **directory** you set yourself in with the command CD.

Current drive
The disk drive that is inserted and declared to be current. The default is SYS:.

Current line
Either the line that EDIT has in its hand at any time or the line pointed at by the cursor in ED.

Current string alteration command
An instruction that changes the current string.

Delimiter characters
Characters used at the beginning and end of a **character string**; that is, characters that define the limits of the string.

Destination file
File being written to.

Device name
Unique name given to a device (for example, DF0: - floppy drive 0).

Directory
A collection of **files**.

Editing commands
Commands input from the keyboard that control an editing session.

Extended mode
Commands appear on the command line and are not executed until you finish the command line.

File
A collection of related data.

Filename
A name given to a **file** for identification purposes.

Immediate mode
Commands are executed immediately.

Keyword
Arguments to **commands** that must be stated explicitly.

Line windows
Parts of line for EDIT to execute subsequent **commands** on.

Memory
This is sometimes known as store and is where a computer stores its data and instructions.

Multi-processing
The execution of two or more **processes** at the same time, rapidly switching from one to the other according to a strict order of priority.

Output queue
Buffer in memory holding data before being written out to **file**.

Priority
The relative importance of a **process**.

Process
A job requested by the operating system or the user; see also Task

Qualifiers
Characters that specify additional conditions for the context in string

Qualified string
A string preceded by one or more qualifiers.

Queue
- see **Output queue**.

Root directory
The top level in the filing system. **Files** and **directories** within the root directory have their names preceded by a colon (:).

Sequential files
A file that can be accessed at any point by starting at the beginning and scanning sequentially until the point is reached.

Source file
File being read from.

Syntax
The format or 'grammar' you use for giving a command.

Task
A task is another name for a process. Tasks are numbered from 1 onwards: task 1 is the initial CLI task, task 2 is the debug task, and so on. To select the next available task, press CTRL-P.

Terminal handler
A process handling input and output from the terminal or console.

Volume name
The unique name associated with a disk.

Wild card
Symbols used to match any pattern.

Tripos User's Reference Manual

This manual refers to Issue 5, May 1986

Printed in the U.K

DISCLAIMER

THIS PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT THE DEVELOPER OR METACOMCO PLC OR ITS AFFILIATED DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, METACOMCO PLC OR ITS AFFILIATED COMPANIES DO NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK. IN NO EVENT WILL METACOMCO PLC OR ITS AFFILIATED COMPANIES BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Tripos User's Reference Manual

Issue 5 (May 1986)

Chapter 1: Tripos Commands

This chapter describes the Tripos commands that are provided as standard. These commands fall into several categories: file utilities, CLI control, command sequence control, system and storage management, and programming tools. Each command is listed in alphabetical order with a definition of its format, template, and purpose; a full specification of the command and an example of its use is also provided.

The chapter starts with a list of unfamiliar terminology. At the end of the chapter there is a Quick Reference Card that lists all the commands according to the above categories (that is, by function).

Table of Contents

1.1 Tripos Commands

Unfamiliar Terminology

In this manual you could find some terms that you have not seen before. The list below includes some common terms that are confusing if you are unfamiliar with them.

Boot startup. It comes from the expression "pulling yourself up by your bootstraps."

Default initial setting or, in other words, what happens if you do nothing. So that, in this manual, 'default' is used to mean 'in absence of something else.'

Device name part of a name that precedes the colon (:), for example, DF0:, SER:, AUX:, and so forth.

File handle an internal Tripos value that represents an open file or device.

Logical device a name you can give to a directory with ASSIGN that you can then use as a device name.

Object code binary output from an assembler or compiler, and binary input to a linker.

Reboot restart.

Stream an open file or device that is associated with a file handle. For example, the input stream could be from a file and the output stream could be to the console device.

System disk a disk containing the commands.

Volume name a name you give to a physical disk.

;

Format: [<command>];[<comment>]

Template: "command";"comment"

Purpose: To add comments to command lines.

Specification:

The CLI ignores everything after the semicolon (;).

Examples:

```
; This line is only a comment
```

ignores the part of the line containing "This line is only a comment."

```
copy <file> to par: ; print the file
```

copies the file to the printer, but ignores the comment "print the file."

See also: C

> <

Format: <command> [>[TO] <out file>]
[<[FROM] <in file>][<arg>*]

Template: "command" >"TO" <"FROM" "args"

Purpose: To direct command input and output.

Specification:

You use the symbols > and < to direct the output and input of a command. The direction of the point of the angle bracket indicates the direction of information flow. You can use these symbols to change where any command reads input or writes output. The output from a command usually goes to the screen. However, if you type a > symbol after a command and before a filename, the command writes the output to that file instead. Similarly, if you type the < symbol before a filename, the command reads from that file instead of from the keyboard.

You do not have to specify both the TO and FROM directions and files. The existence and number of "args" depends on the command you used. Redirection only happens for the command you specified. Tripos reverts to the initial or 'default' input and output (that is, the keyboard and screen) afterwards.

Examples:

```
DATE > diary_dates
```

writes the output of the DATE command (that is, today's date and time) to the file 'diary__dates'.

```
my_program < my_input
```

tells my__program to acccept input from my__input instead of from the keyboard.

```
LIST > temp
SORT temp TO *
```

produces a sorted list of files and displays them on the screen.

The following sequence:

```
ECHO > 2nd.date 02-jan-78
DATE < 2nd.date ?
DELETE 2nd.date
```

creates a file called 2nd.date that contains the text "02-jan-78<linefeed>". Next it uses this file as input to the command DATE. Note that the '?' is necessary for DATE to accept input from the standard input, rather than the command line. Finally, as you no longer need the file, the DELETE command deletes 2nd.date.

ALINK

Format: ALINK [[FROM|ROOT] <filename>
[,<filename>* | + <filename*]][TO
<name>][WITH <name>][LIBRARY|LIB
<name>][MAP <map>][XREF <name>]
[WIDTH <n>][SMALL]

Template: ALINK "FROM=ROOT,TO/K,WITH/K,VER/K,
LIBRARY=LIB/K,MAP/K,XREF/K,
WIDTH/K,SMALL/S"

Purpose: To link together sections of code into an executable file.

Specification:

ALINK instructs Tripos to link files together. It also handles automatic library references and builds overlay files. The output from ALINK is a file loaded by the loader and run under the overlay supervisor, if required.

For details and a full specification of the ALINK command, see Chapter 5 of the *Tripos Programmer's Reference Manual*.

Examples:

```
ALINK a+b+c TO output
```

links the files 'a', 'b' and 'c', producing an output file 'output'.

ASSEM

Format: ASSEM [PROG|FROM] <prog> [TO <code>]
[VER <ver>] [LIST <listing>]
[EQU <equates file>] [OPT <opt>]
[INC <dirlist>] [OBJ <object
module file>]

Template: ASSEM "PROG=FROM/A,TO/K,VER/K,
LIST/K,HDR/K,EQU/K,OPT/K,
INC/K,OBJ/K"

Purpose: To assemble a program in MC68000 assembly language.

Specification:

ASSEM assembles programs in MC68000 assembly language. See Chapter 4 of the *Tripos Programmer's Reference Manual* for details.

PROG	is the source file.
TO	is the object file (that is, binary output from the assembler)
VER	is the file for messages (unless you specify VER, messages go to the terminal)
LIST	is the listing file.
OPT	specifies options to the assembler.
HDR	is a header file which can be read as if inserted at the front of the source (like INCLUDE in the source itself).
INC	sets up a list of directories to be searched for included files.
EQU	is the file that receives the 'equates' directive (EQU) assignments from your source. You use EQU to generate a header file containing these directives.

The options you can specify with OPT are as follows:

S	produce a symbol table dump as part of the object file.
D	inhibit the dumping of local lables as part of a symbol dump.
C	ignore the distinction between upper and lower case in labels.
X	produce a cross-reference file.
L	produce a listing file with the default suffix.
N	inhibit production of object files.

Examples:

```
ASSEM prog.asm TO prog.obj
```

assembles the source program in 'prog.asm', placing the result in the file 'prog.obj'. It writes any error messages to the terminal, but does not produce an assembly listing.

```
ASSEM prog.asm TO prog.obj HDR slib LIST prog-list
```

assembles the same program to the same output, but includes the file 'slib' in the assembly, and places an assembly listing in the file 'prog-list'.

ASSIGN

Format: ASSIGN [[<name>] <dir>] [LIST]

Template: ASSIGN "NAME,DIR,LIST/S"

Purpose: To assign a logical device name to a filing system directory or to examine current device names.

Specification:

NAME is the logical device name given to the directory specified by DIR.

ASSIGN <name> deletes the logical device name given (that is, it removes the assignment of <name>).

ASSIGN, or ASSIGN LIST, lists all current assignments.

When you use ASSIGN, you must ensure that a disk is in the drive: ASSIGN makes an assignment to a disk, not to a drive.

Note: Restarting the computer removes your ASSIGNments.

Examples:

```
ASSIGN sources: :new/work
```

Sets up the logical device name 'sources' to the directory ':new/work'. Then to gain access to files in ':new/work', you can use the logical device name 'sources', as in

```
TYPE sources:xyz
```

which displays the file ':new/work/xyz'.

```
ASSIGN LIST
```

lists the current logical device names in use. (Used to change the standard assignments such as C:, L:, SYS:, etc.)

BREAK

Format: BREAK <task> [ALL] [C] [D] [E] [F]

Template: BREAK "TASK/A,ALL/S,C/S,D/S,E/S,F/S"

Purpose: To set attention flags in the given task (process).

Specification:

BREAK sets the specified attention flags in the task. C sets the CTRL-C flag, D sets the CTRL-D flag, and so on. ALL sets all the flags from CTRL-C through CTRL-F. If you just specify <task>, Tripos sets the CTRL-C flag.

Note: It is the programmer's responsibility to detect and act on these flags being set. Tripos doesn't actually stop or remove a command.

Examples:

```
BREAK 7
```

sets the CTRL-C attention flag of task 7. This is identical to selecting task 7 and pressing CTRL-C.

```
BREAK 5 D
```

sets the CTRL-D attention flag of task 5.

```
BREAK 3 D E
```

sets both CTRL-D and CTRL-E.

C

Format: C <commandfile> [<arg>*]

Template: C "command-file","args"

Purpose: To execute a file of commands with argument substitution.

Specification:

You normally use C to save typing. The command-file contains commands executed by the Command Line Interface. Tripos executes these commands one at a time, just as though you had typed them at the keyboard.

You can also use C to perform parameter (that is, value) substitution, where you can give certain names as parameters. Before the command file is executed, Tripos checks the parameter names with those you've given after the C command. If any match, Tripos uses the values you specified instead of the parameter name. Parameters may have values specified that Tripos uses if you do not explicitly set the parameter. If you have not specified a parameter, and if there is no default, then the value of the parameter is empty and nothing is substituted for it.

To use parameter substitution, you give directives to the C command. To indicate these, you start a line with a special character, which is initially a period or 'dot' (.). The directives are as follows:

.KEY		Argument template, used to specify the format of the arguments
.K		Argument template, identical to .KEY
.DOT	ch	Change dot character (initially '.') to ch
.BRA	ch	Change bra character (initially '<') to ch
.KET	ch	Change ket character (initially '>') to ch
.DOLLAR	ch	Change default-char (initially '$') to ch
.DOL	ch	Equivalent to .DOLLAR
.DEF	keyword value	Give default to parameter
.<space>		Comment line
.<newline>		Blank comment line

Before execution, Tripos scans the contents of the file for any items enclosed by BRA and KET characters ('<' and '>'). Such items may consist of a keyword or a keyword and a default value for Tripos to use if you have left the keyword unset. (To separate the keyword and the default, if there is one, you type a dollar sign '$'). Thus, Tripos replaces <ANIMAL> with the value you associated with the keyword ANIMAL, while it replaces <ANIMAL$WOMBAT> with the value of ANIMAL if it has one, and otherwise it defaults to WOMBAT.

A file can only use the dot commands if the first line has a dot command on it. The CLI looks at the first line. If it starts with a dot command, for example, a comment (.<space>txt) then the CLI scans the file looking for parameter substitution and builds a temporary file in the :T directory. If the file doesn't start with a dot command, then it is assumed that there are NO dot commands in the file, which also means no parameter substitution is performed. For the no-dot case, the CLI starts executing the file directly without having to copy it to :T. Note that you can still embed comments in a command-file by using the CLI's comment character, the semicolon (;). If you don't need parameter substitution and dot commands, don't use '.' as a comment. Not using '.' saves you extra accesses to the disk for the temporary file.

Tripos provides a number of commands that are only useful in command sequence files. These include IF, SKIP, LAB, and QUIT. These can be nested in a command file.

Note that you can also nest C files. That is, you can have a command file that contains C commands.

To stop the execution of a command file, you press CTRL-D. If you are nesting command files; that is, if one command file calls another, you can stop the entire set of C commands by pressing CTRL-C. CTRL-D only stops the current command file from executing.

Examples:

Assume the file 'list' contains the following:

```
.k filename/a
run copy <filename> to par:+
echo "Printing of <filename> done"
```

Then the following command

```
C list test/prg
```

acts as though you had typed the following commands at the keyboard.

```
RUN copy test/prg to par:+
ECHO "Printing of test/prg done"
```

Another example, "display", uses more of the features described above:

```
.key name/a
IF EXISTS <name>
TYPE <name> OPT n
. If the file given is on the current
. directory, type it with line numbers.
ELSE
ECHO "<name> is not on this directory"
ENDIF
.

RUN C display work/prg2
```

should display the file work/prg2 with line numbers on the terminal if it exists on the current directory. If the file is not there, the screen displays the following message:

```
work/prg2 is not on this directory.
```

See also: ;, IF, SKIP, FAILAT, LAB, ECHO, RUN, QUIT

CD

Format: CD [<dir>]

Template: CD "DIR"

Purpose: To set or change a current directory or drive.

Specification:

CD with no parameters displays the name of the current directory. In the format list above, <dir> indicates a new current directory (that is, one in which unqualified filenames are looked up). If the directory you specify is not on the current drive, then CD also changes the current drive.

To change the current directory to the directory that owns the current one (if one exists), type CD followed by a single slash (/). Thus CD / moves the current directory one level up in the hierarchy unless the current directory is a root directory (that is, the top level in the filing system). Multiple slashes are allowed; each slash refers to an additional level above.

Examples:

```
CD df1:work
```

sets the current directory to 'work' on disk 'df1', and sets the current drive to 'df1'.

```
CD SYS:COM/BASIC
CD /
```

sets the current directory to 'SYS:COM'.

CONSOLE

Format: CONSOLE [WIDTH <integer>] [TAB ON|OFF] [TASK <key>|OFF] [PAGE ON|OFF][LENGTH <integer>] [TABSTOP <integer>] [AUTONL ON|OFF] [IFC ON|OFF]

Template: CONSOLE "WIDTH/K,TAB/K,TASK/K, PAGE/K,LENGTH/K,TABSTOP/K,AUTONL/K IFC/K"

Purpose: To set the console characteristics of the console handler used by the current CLI.

Specification:

WIDTH Sets the width of the console. The value of WIDTH must be a valid positive integer. The initial value is 80.

TAB Controls tabular spacing. TAB can have the value ON or OFF. TAB ON ensures that tab characters (HT) are expanded to spaces by the console handler during both input and output. TAB OFF ensures they are untouched. TAB is initially ON.

TASK Allows you to set the task-changing control key (default CTRL-P - think of P for Process, the other name for task). For example, TASK T (as in T for Task) sets the control to CTRL-T. TASK OFF disables task-changing. Take care in your choice of key.

PAGE Specifies if output is to be in page mode. PAGE can have the value ON or OFF. If PAGE mode is ON, the system automatically waits at the end of each page (screen-full) of output; the control code CTRL-Q must be given for the display to continue. The default mode is OFF, where no automatic page waits are performed, although output can be halted at any stage by pressing CTRL-S or starting an input line.

LENGTH Sets the console length to be used in PAGE mode. This must be a positive integer giving the number of lines on the screen. The default length is 24.

TABSTOP Specifies the number of spaces the horizontal tab is to represent. By default, the console handler expands a tab to 3 spaces.

AUTONL Indicates whether the console handler is to insert a newline when the line printed hits the specified terminal width. AUTOLN can have the value ON or OFF. Some terminals provide for an extra half cursor position so that characters can be written in the final position; in this case the default ON is suitable. Other terminals may automatically perform a newline when the final character position has been filled; in this case AUTONL should be set OFF.

IFC Stands for Input Flow Control. 'Input' is input to the console handler. This can be input from the keyboard; however, the rate of input flow from the keyboard to the console handler is unlikely to be too fast to handle. You usually use IFC ON to inhibit input flow from another computer that is running as a separate 'terminal'. IFC allows the console handler to control the rate of input flow from this 'terminal' by switching the flow off and then on again. The console handler inhibits input by sending XOFF (CTRL-S); it then re-enables input when it is ready by sending XON (CTRL-Q). The default is OFF.

If no parameters are given, the current state of all the options is printed.

Examples:

```
CONSOLE
```

prints the current option settings.

```
CONSOLE PAGE ON LENGTH 20
```

turns page mode on and sets the screen display to 20 lines. This ensures the console stops and waits for a CTRL-Q after displaying 20 lines of output.

COPY

Format: COPY [[FROM] <name>] [TO <name>] [ALL] [QUIET]

Template: COPY "FROM,TO/A,ALL/S,QUIET/S"

Purpose: To copy a file or directory from one place to another.

Specification:

COPY places a copy of the file or directory in the file or directory specified as TO. The previous contents of TO, if any, are lost. If you specify FROM as a file and TO as a directory, COPY creates a copy of the FROM file in the TO directory (that is, the contents of <file> are copied to <dir>/<file>).

If you specify a directory name as FROM, COPY copies all the files in the FROM directory to the TO directory. If you do not specify the FROM directory, Tripos uses the current directory. The TO directory must exist for COPY to work; it is not created by COPY.

If you specify ALL, COPY also copies the files in any subdirectories. In this case, it automatically creates subdirectories in the TO directory, as required. The name of the current file being copied is displayed on the screen as it happens unless you give the QUIET switch.

You can also specify the source directory as a pattern. In this case, Tripos copies any files that match the pattern. See the command LIST for a full description of patterns. You may specify directory levels as well as files as patterns.

Examples:

```
COPY file1 TO :work/file2
```

copies 'file1' in the current directory to 'file2' in the directory ':work'.

```
COPY TO df1:backup
```

copies all the files in the current directory to 'df1:backup'. It does not copy any subdirectories, and DF1:backup must already exist.

```
COPY df0: to df1: ALL QUIET
```

makes a logical copy of disk 'df0' on disk 'df1.' No consideration is given to filenames. All files an sub-directories on the disk are copied.

```
COPY test-#? to df1:xyz
```

copies all files in the current directory that start 'test-' to the directory xyz on the disk 'df1', assuming that 'xyz' already exists. (For an explanation of patterns, such as '#?', see the command LIST in this chapter.)

```
COPY test_file to PAR:
```

copies the file 'test_file' to your printer.

```
COPY DF0:?/#? TO DF1: ALL
```

copies every file in any 1 character subdirectory of DF0: to the root directory of DF1:.

See also: JOIN

DATE

Format: DATE [<date>] [<time>] [TO|VER <name>]

Template: DATE "DATE,TIME,TO=VER/K"

Purpose: To display or set the system date or time.

Specification:

DATE with no parameter displays the currently set system date and time. This includes the day of the week. Time is displayed using a 24-hour clock.

DATE <date> sets the date. The form of <date> is DD-MMM-YY. If the date is already set, you can reset it by specifying a day name (this sets the date forward to that day) or by specifying 'tomorrow' or 'yesterday'.

DATE <time> sets the time. The form of <time> is HH:MM (for Hours and Minutes). You should use leading zeros when necessary. Note that, if you use a colon (:), Tripos recognizes that you have specified the time rather the date. That is to say, you can set both the date and the time or either and in any order because DATE only refers to the time when you use the form HH:MM.

If you do not set the date, the restart disk validation process sets the system date to the date of the most recently created file.

To specify the destination of the verification, you use the equivalent keywords TO and VER. The destination is the terminal unless you specify otherwise.

Note: If you type DATE before the restart validation has completed, the time is displayed as unset. To set the time, you can either use DATE or just wait until the validation process is finished.

Examples:

```
DATE
```

displays the current date.

```
DATE 06-Sep-82
```

sets the date to the 6th of September 1982. The time is not reset.

```
DATE tomorrow
```

resets the date to one day ahead.

```
DATE TO fred
```

sends the current date to the file "fred".

```
DATE 10:50
```

sets the current time to ten 'til eleven.

```
DATE 23:00
```

sets the current time to 11:00 p.m.

```
DATE 01-JAN-02
```

sets the date to January 1st, 2002. (The earliest date you can set is 02-JAN-78.)

DELETE

Format: DELETE <name> [<name>*][ALL][Q|QUIET]

Template: DELETE ",,,,,,,,,,ALL/S,Q=QUIET/S"

Purpose: To delete up to ten files or directories.

Specification:

DELETE attempts to delete each file you specify. If it cannot delete a file, the screen displays a message, and Tripos attempts to delete the next file in the list. You may not delete a directory if it contains any files.

You can also use a pattern to specify the filename. See the description of the command LIST for full details of patterns. The pattern may specify directory levels as well as filenames. In this case, all files that match the pattern are deleted.

If you specify ALL with a directory name, DELETE will delete that directory and all subdirectories and files within that directory and its subdirectories.

Unless you specify the switch QUIET (or use the alternative, Q), the name of the file being deleted appears on the screen as it happens.

Examples:

```
DELETE old-file
```

deletes the file 'old-file'.

```
DELETE work/prog1 work/prog2 work
```

deletes the files 'prog1' and 'prog2' in the directory 'work', and then deletes the directory 'work'.

```
DELETE t#?/#?(1|2)
```

deletes all the files that end in '1' or '2' in directories that start with 't'. (For an explanation of patterns, such as '#?', see the command LIST later in this chapter.)

```
DELETE DF1:#? ALL
```

deletes all the files on DF1:.

See also: DIR (I-DEL option)

DIR

Format: DIR [<name>] [OPT A|I|D]

Template: DIR "DIR,OPT/K"

Purpose: To provide a display of the files in a directory in sorted order. DIR can also include the files in sub-directories. You can use DIR in interactive mode.

Specification:

DIR alone shows the files in the current directory. DIR followed by a directory provides the files in that directory. The form of the display is first any subdirectories, followed by a sorted list of the files in two columns. If you want to know if a file exists type LIST filename.

Typing DIR filename, where filename is a file which exists results in the computer responding with: "filename is not a directory."

To pass options to DIR, use the OPT keyword. Use the A option to include any subdirectories below the specified one in the list. Each sublist of files is indented.

To list only the directory names use the D option.

The I option specifies that DIR is to run in interactive mode. In this case, the files and directories are displayed with a question mark following each name. Press RETURN to display the next name in the list. To quit the program, type Q. To go back to the previous directory level or to stop (if at the level of the initial directory), type B.

Any combination of the three options may be used.

If the name displayed is that of a directory, type E to enter that directory and display the files and subdirectories. Use E and B to select different levels. Typing the command DEL (that is, typing the three letters D E L, not pressing the DEL key) can be used to delete a directory, but this only works if the directory is empty.

If the name is that of a file, typing DEL deletes the file, or typing T Types (that is, displays) the file at the screen. In the last case, press CTRL-C to stop it 'typing' and return to interactive mode.

To find the possible responses to an interactive request, type ?.

Examples:

```
DIR
```

provides a list of files in current directory.

```
DIR df0: OPT a
```

lists the entire directory structure of the disk 'df0'.

DISKCOPY

Format: DISKCOPY [FROM] <disk> TO <disk> [NAME <name>]

Template: DISKCOPY "FROM/A,TO/A/K,NAME/K"

Purpose: To copy the contents of one floppy disk to another.

Specification:

DISKCOPY makes a copy of the entire contents of the disk you specified as FROM, overwriting the previous contents of the entire disk you specified as TO. If you use a new, unformatted disk as the TO disk, you must format it first. You normally use the command to produce backup floppy disks.

Once you have given the command, Tripos prompts you to insert the correct disks. At this point, you insert the correct source (FROM) and destination (TO) disks.

You can use the command to copy any Tripos disk to another, but the source and destination disks must be identical in size and structure. To copy information between different sized disks, you use COPY.

You can also use the command to copy a floppy disk using a single floppy drive. If you specify the source and destination as the same device, then the program reads in as much of the source disk into memory as possible. It then prompts you to place the destination disk in the drive and then copies the information from memory onto the destination disk. This sequence is repeated as many times as required.

If you do not specify a new name for your disk, DISKCOPY creates a new disk with the same name as the old one. However, Tripos can tell the difference between two disks with the same name because every disk is associated with the date and time of its creation. DISKCOPY gives the new disk the current system date as its creation date and time.

Note: If you only have one disk drive, you can use COPY to RAM: to copy part of a disk.

Examples:

```
DISKCOPY FROM df0: TO df1:
```

makes a backup copy of the disk 'df0' onto disk 'df1'.

```
DISKCOPY FROM df0: TO df0:
```

makes a backup copy of the disk in drive 'df0' using only a single drive.

See also: COPY

DISKDOCTOR

Format: DISKDOCTOR [DRIVE] <name>

Template: DISKDOCTOR "DRIVE/A"

Purpose: To restore a corrupt disk.

Specification:

DISKDOCTOR tries to restore to health a disk that has previously failed to validate. When you specify DRIVE DISKDOCTOR prompts for the disk to be inserted. This disk must be in write-enabled state.

DISKDOCTOR scans the disk looking for unreadable blocks, and takes the following actions on finding a bad block:

1. If the block is a **directory block**, it places all files and subdirectories in that directory block into the root directory of the disk. This may result in two files in the root having the same name, but you can resolve this by renaming one of them.

2. If the block is a **file header block**, it warns you that a file header block in a particular directory is unreadable. (DISKDOCTOR can't tell you the name of the file because it can't read the header block.) It then deletes the file; this may mean that you also lose other files as a result.

3. If the block is a **file data block**, it warns you that part of the file is unreadable. However, DISKDOCTOR will not delete the file. When DISKDOCTOR has finished, the disk validator will automatically be invoked, hopefully resulting in a perfectly healthy disk, although files which are unreadable cannot be copied.

After using DISKDOCTOR you should copy all the data you want (or can) to a fresh disk. You should then reformat the corrupted disk. If the disk fails to format, you should discard it.

ECHO

Format: ECHO <string>

Template: ECHO ""

Purpose: To display the argument given.

Specification:

ECHO writes <string> to the current output stream (which can be a file or a device). This is normally only useful within a command sequence or as part of a RUN command. If you give the argument incorrectly, an error is displayed.

Examples:

```
RUN COPY :work/prog to df1:work ALL QUIET +
ECHO "Copy finished"
```

creates a new CLI to copy the specified directory as a background task. On completion, the message "Copy finished" appears.

To copy 2 files to the RAM disk and back, type

```
C mycopy
```

after entering the following command file:

```
ECHO "Starting 'MYCOPY' command file"
COPY DF1:ABC TO RAM:ABC
COPY DF1:XYZ TO RAM:XYZ
ECHO "Remove the diskette in DF1:"
ECHO "Insert the new diskette in DF1:"
WAIT 10 SECS
COPY RAM:ABC TO DF1:ABC
COPY RAM:XYZ TO DF1:ABC
ECHO "Done"
```

ED

Format:

ED [FROM] <name> [SIZE <n>]

Template:

ED "FROM/A,SIZE"

Purpose: To edit text files.

Specification:

ED is a screen editor. You can use ED as an alternative to the line editor EDIT. The file you specify as FROM is read into memory, then ED accepts your editing instructions. If FROM filename does not exist, Tripos creates a new file.

Because the file is read into memory, there is a limit to the size of file you can edit with ED. Unless you specify otherwise, workspace size is determined by the size of the file. If you wish to edit a file and insert a large amount of extra text, you may wish to alter the workspace size. To alter the workspace, you specify a suitable value after the SIZE keyword.

There is a full specification of ED in Chapter 2.

Examples:

```
ED work/prog
```

edits the file 'work/prog', assuming it exists; otherwise, ED creates the file.

```
ED huge-file SIZE 50000
```

edits a very large file 'huge-file', using a workspace of 50,000 bytes.

EDIT

Format: EDIT [FROM] <name> [[TO] <name>][WITH <name>][VER <name>][OPT <option>]

Template: EDIT "FROM/A,TO,WITH/K,VER/K,OPT/K"

Purpose: To edit text files.

Specification:

EDIT is a line editor (that is, it edits a sequential file line by line). If you specify TO, EDIT copies from file FROM to file TO. Once you have completed the editing, the file TO contains the edited result, and the file FROM is unchanged. If you do not specify TO, then EDIT writes the edited text to a temporary file. If you give the EDIT commands Q or W, then EDIT renames this temporary file FROM, having first saved the old version of FROM in the file ':t/edit-backup'. If you give the EDIT command STOP, then EDIT makes no change to the file FROM.

EDIT reads commands from the current input stream, or from a WITH file if it specified.

EDIT sends editor messages and verification output to the file you specify with VER. If you omit VER, the terminal is used instead.

OPT specifies options: Pn sets the maximum number of previous lines to n; Wn sets the maximum line width. The initial setting is P40W120.

Note: You cannot use the < and > symbols to redirect input and output when you call EDIT.

See Chapter 3 for a full specification of EDIT.

Examples:

```
EDIT work/prog
```

edits the file 'work/prog'. When editing is complete, EDIT saves the old version of 'work/prog' in ':t/edit-backup'.

```
EDIT work/prog TO work/newprog
```

edits the file 'work/prog', placing the edited result in the file 'work/newprog'.

```
EDIT work/prog WITH edits/0 VER nil:
```

edits the file 'work/prog' with the edit commands stored in the file 'edits/0'. Verification output from EDIT is sent to the dummy device 'nil:'.

ENDCLI

Format: ENDCLI

Template: ENDCLI

Purpose: To end an interactive CLI task.

Specification:

ENDCLI removes the current CLI.

You shouldn't use ENDCLI except on a CLI created by the NEWCLI command. If the initial CLI (task 1) is ended, and no other has been set up by the NEWCLI command, then the effect is to terminate the Tripos session.

Note that there are no arguments to the ENDCLI command, and no check for invalid arguments.

Note: Do not experiment with ENDCLI before you've used NEWCLI. Using ENDCLI on the initial CLI always pulls the rug out from under you by terminating that CLI, and ending the last CLI gives you no way of creating a new one.

Examples:

```
NEWCLI
[Select the new CLI by pressing CTRL-P until it is selected]
LIST
ENDCLI
```

opens a new CLI, lists the directory, and closes the CLI again.

FAILAT

Format: FAILAT <n>

Template: FAILAT "RCLIM"

Purpose: To instruct a command sequence to fail if a command returns an error code greater than or equal to the number specified.

Specification:

Commands indicate that they have failed by setting a return code. A non-zero return code indicates that the command has found an error. A return code >= the fail limit terminates a sequence of non-interactive commands (that is, commands specified after RUN or in a C file). The return code indicates how serious the error was; it is usually 5, 10 or 20.

You may use the FAILAT command to alter this fail level from its initial value of 10. If you increase the level, you indicate that certain classes of error should not be regarded as fatal, and that execution of subsequent commands may proceed after an error. <n> should be a positive number. The fail level is reset to the initial value of 10 on exit from the command sequence.

You must use FAILAT before commands such as IF to test to see if a command has failed; otherwise, the command sequence terminates before executing the IF command.

If you omit the argument, the current value of the fail level is displayed.

Examples:

```
FAILAT 25
```

ends the command sequence if a command stops with a return code >=25.

See also: IF, C, RUN, QUIT

FAULT

Format: FAULT[<n>*]

Template: FAULT ",,,,,,,,"

Purpose: To display the messages corresponding to the fault codes you supply.

Specification:

Tripos looks up the numbers and displays the corresponding messages. Up to ten messages may be displayed.

Examples:

```
FAULT 222
```

displays the message for fault 222.

```
FAULT 221 103 121 218
```

displays the messages for faults 221, 103, 121, and 218.

FILENOTE

Format: FILENOTE [FILE] <file> COMMENT <string>

Template: FILENOTE "FILE/A,COMMENT/K"

Purpose: To attach a comment or a note to a file.

Specification:

FILENOTE assigns a comment to a specified file.

The keyword COMMENT introduces an optional comment of up to 80 characters. A comment may be more than one word (that is, contain spaces between characters). In this case, you must enclose the comment within double quotes (").

A comment is associated with a particular file. When you examine the file with the command LIST, the comment appears on the line below:

```
prog                30 rwed  Today  11:07:33
 : version 3.2 - 23-mar-86
```

When you create a new file, it does not normally have a comment. If you overwrite an existing file that has a comment, then the comment is retained even though the contents of the file has changed. The command COPY copies a file. If a file with a comment is copied, the new file does not have the comment from the original attached to it although the destination file may have a comment which is retained.

Examples:

```
FILENOTE prog2  COMMENT "Ver 3.3 26-mar-86"
```

attaches the comment "Ver 3.3 26-mar-86" to program 2.

FORMAT

Format: FORMAT DRIVE <drivename> NAME <string>

Template: FORMAT "DRIVE/A/K,NAME/A/K"

Purpose: To format and initialize a new floppy disk.

Specification:

The program formats a new floppy disk in the method required for Tripos. Once the disk is formatted, it is initialized and assigned the name you specify. The DRIVE and NAME keywords must be given. DRIVE specifies the drive name (for example, DF0:). You can type any string after NAME, but if you use spaces, you must enclose the whole string in double quotes (").

WARNING: FORMAT formats and initializes a disk as an empty disk. If you use a disk that is not empty, you'll lose the previous contents of the disk.

The name assigned should be unique. It may be 1 to 30 characters in length and composed of one or more words separated by spaces. If the name is more than one word, you should enclose it in double quotes.

Examples:

```
FORMAT DRIVE df0: NAME "Work disk"
```

formats and initializes the disk in drive 'df0' with the name "Work disk".

See also: DISKCOPY, RELABEL

IF

Format: IF [NOT] [WARN] [ERROR] [FAIL] [<str> EQ <str>] [EXISTS <name>]

Template: IF "NOT/S,WARN/S,ERROR/S,FAIL/S,, EQ/K,EXISTS/K"

Purpose: To allow conditionals within command sequences.

Specification:

You can only use this command in a C command file. If one or more of the specified conditions is satisfied, IF carries out all the following commands until it finds a corresponding ENDIF or ELSE command; otherwise, if the conditions are not satisfied, it carries out whatever follows a corresponding ELSE command. (ENDIF and ELSE are only useful in command sequences containing IF.) ENDIF terminates an IF command; ELSE provides an alternative if the IF conditions fail. Note that the conditions and commands in an IF and ELSE command can span more than one line before their corresponding ENDIF.

The following table shows some of the ways you can use the IF, ELSE, and ENDIF commands:

IF <cond>	IF <cond>	IF <cond>
<command>	<command>	<command>
ENDIF	ELSE	IF <cond>
	<command>	<command>
	ENDIF	ENDIF
		ENDIF

Note that ELSE is optional and that nested IFs jump to the nearest ENDIF.

ERROR is only available if you set FAILAT to greater than 10. Similarly, FAIL is only available if you set FAILAT to greater than 20.

Keyword	Function
NOT	reverses the result.
WARN	satisfied if previous return code >= 5.
ERROR	" " " " " >= 10.
FAIL	" " " " " >= 20.
<a> EQ <b>	satisfied if the text of a and b is identical (disregarding case).
EXISTS <file>	satisfied if the file exists

You can use IF EQ to detect an unset parameter in a command file by using the form

```
IF <a> EQ ""
```

Examples:

```
IF EXISTS work/prog
TYPE work/prog
ELSE
ECHO "file not found"
ENDIF
```

If the file 'work/prog' exists, then Tripos displays it. Otherwise, Tripos displays the message "file not found" and executes the next command in the command sequence.

```
IF ERROR
SKIP errlab
ENDIF
```

If the previous command stopped with a return code >= 10, then Tripos skips the command sequence until you define a label 'errlab' with the LAB command.

```
IF ERROR
IF EXISTS fred
ECHO "An error occurred; 'fred' exists."
ENDIF
ENDIF
```

See also: FAILAT, SKIP, LAB, C, QUIT

INFO

Format: INFO

Template: INFO

Purpose: To give information about the filing system.

Specification:

The command displays a line of information about each disk unit. This includes the maximum size of the disk, the current used and free space, the number of soft disk errors that have occurred, and the status of the disk.

Examples:

```
INFO

Unit  Size  Used  Free  Full  Errs  Status      Name
DF1:  880K     2  1756    0%  0     Read/Write  Test6

Volumes available:
Test-6 [Mounted]
Tripos CLI V27.5 4-Jul-85 [Mounted]
```

INSTALL

Format:

INSTALL <file> [VERSION A|B|C|D]

Template:

INSTALL "FILE/A,VERSION/K"

Purpose: To install a new Tripos system image onto a disk

Specification:

The filename you specify as <file> is identified as a bootstrap Tripos system image for the disk on which it resides. The file must be the the output from SYSLINK. (See the *Tripos Technical Reference Manual* for a description of SYSLINK.) You may install up to 4 different bootstraps on a single disk; that is, versions A, B, C, and D. If you do not specify VERSION, the file is installed as version A. A particular version is chosen as part of the bootstrap procedure.

Note: Take care always to keep at least one working version of a Tripos bootstrap.

Examples:

```
INSTALL tripos
```

installs the file 'tripos' as the system image file for version A on the current disk.

```
INSTALL test-tripos VERSION B
```

installs a test version of Tripos as version B.

JOIN

Format: JOIN <name> <name> [<name>*] AS <name>

Template: JOIN ",,,,,,,,,,,,,,,AS/A/K"

Purpose: To concatenate up to 15 files to form a new file.

Specification:

Tripos copies the specified files in the order you give into the new file. Note that the new file cannot have the same name as any of the input files.

Examples:

```
JOIN part1 part2 AS textfile
```

joins the two files together, placing the result in 'textfile'. The two original files remain unchanged, while 'textfile' contains a copy of 'part1' and a copy of 'part2'.

LAB

Format: LAB <string>

Template: LAB <text>

Purpose: To implement labels in command sequence files.

Specification:

The command ignores any parameters you give. Use LAB to define a label 'text' that is looked for by the command SKIP.

Examples:

```
LAB errlab
```

defines the label 'errlab' to which SKIP may jump.

See also: SKIP, IF, C

LIST

Format: LIST [[DIR] <dir>] [P|PAT <pat>] [KEYS] [DATES] [NODATES] [TO <name>] [S <str>] [SINCE <date>] [UPTO <date>] [QUICK]

Template: LIST "DIR,P=PAT/K,KEYS/S,DATES/S, NODATES/S,TO/K,S/K,SINCE/K, UPTO/K,QUICK/S"

Purpose: To examine and list specified information about a directory or file.

Specification:

If you do not specify a name (the parameter DIR), LIST displays the contents of the current directory. The first parameter LIST accepts is DIR. You have three options. DIR may be a filename, in which case LIST displays the file information for that one file. Secondly DIR may be a directory name. In this case LIST displays file information for files (and other directories) within the specified directory. Lastly, if you omit the DIR parameter, LIST displays information about files and directories within the current directory (for further details on the current directory, see the CD command).

Note: LIST, unlike DIR, does NOT sort the directory before displaying it.

If no other options are specified, LIST displays

```
file_name       size  protection date time
:comment
```

These fields are defined as follows:

file_name: Name of file or directory.

size: The size of the file in bytes. If there is nothing in the file, this field will state "empty". For directories this entry states "dir".

protection: This specifies the access available for this file. rwed indicates Read, Write, Execute, and Delete.

date and time: The file creation date and time.

comment: This is the comment placed on the file using the FILENOTE command. Note that it is preceded with a colon (:).

Options available:

TO This specifies the file (or device) to output the file listing to. If omitted, the output goes to the current CLI.

KEYS displays the block number of each file header or directory.

DATES displays dates in the form DD-MMM-YY (the default is to display, where applicable, a day name in the last week, TODAY or YESTERDAY).

NODATES does not display date and time information.

SINCE <date> displays only files last updated on or after <date>. <date> can be in the form DD-MMM-YY or a day name in the last week (for example, MONDAY) or TODAY or YESTERDAY.

UPTO <date> displays only files last updated on or before <date>.

P <pat> searches for files whose names match <pat>.

S <str> searches for filenames containing substring <str>.

QUICK	just displays the names of files and directories (like the DIR command).

You can specify the range of filenames displayed in two ways. The simplest way is to use the S keyword, which restricts the listing to those files containing the specified substring. To specify a more complicated search expression, use the P or PAT keyword. This is followed by a pattern that matches as described below.

A pattern consists of a number of special characters with special meanings, and any other characters that match themselves.

The special characters are: ' ( ) ? % # |

In order to remove the special effect of these characters, preface them with '. Thus '? matches ? and '' matches '.

?	matches any single character.
%	matches the null string.
#<p>	matches zero or more occurrences of the pattern <p>.
<p1><p2>	matches a sequence of pattern <p1> followed by <p2>.
<p1>|<p2>	matches if either pattern <p1> or pattern <p2> match.
()	groups patterns together.

Thus:

LIST PAT A#BC	matches AC ABC ABBC, and so forth.
LIST PAT A#(B|C)D	matches AD ABD ABCD, and so forth.
LIST PAT A?B	matches AAB ABB ACB, and so forth.
LIST PAT A#?B	matches AB AXXB AZXQB, and so forth.
LIST PAT '?#?'#	matches ?# ?AB# ??##, and so forth
LIST PAT A(B|%)#C	matches A ABC ACCC, and so forth.
LIST PAT #(AB)	matches AB ABAB ABABAB, and so forth.

Examples:

```
LIST
```

displays information about all the files and directories contained in the current directory. For example,

```
File_1
File 2
File.3
:comment
File004
```

notice that File.3 has a comment.

```
LIST work S new
```

displays information about files in the directory 'work' whose names contain the text 'new'. Note that LIST S produces the response: "Args no good for key" because there is an "S" directory. LIST "s" will list this directory's contents.

```
LIST work P new#?(x|y)
```

examines the directory 'work', and displays information about all files that start with the letters 'new' and that end with either 'x' or 'y'.

```
LIST QUICK TO outfile
```

sends just the names, one on each line, to the file 'outfile'. You can then edit the file and insert the command TYPE at the beginning of each line. Then type

```
C outfile
```

to display the files.

See also: DATE, DIR, FILENOTE, PROTECT

MAKEDIR

Format: MAKEDIR <dir>

Template: MAKEDIR "/A"

Purpose: To make a new directory.

Specification:

MAKEDIR creates a directory with the name you specify. The command only creates one directory at a time, so any directories on the path must already exist. The command fails if the directory or a file of the same name already exists in the directory above it in the hierarchy.

Examples:

```
MAKEDIR tests
```

creates a directory 'tests' in the current directory.

```
MAKEDIR df1:xyz
```

creates a directory 'xyz' in the root directory of disk 'df1'.

```
MAKEDIR df1:xyz/abc
```

creates a directory 'abc' in the parent directory 'xyz' on disk 'df1:'. However, 'xyz' must exist for this command to work.

See also: DELETE

MOUNT

Format: MOUNT <devicename>

Template: MOUNT "DEV/A"

Purpose: To mount a new device.

Specification:

The MOUNT command mounts (makes available) a non-standard device. Standard devices (SER:, SYS:, etc) are mounted automatically by the system.

To use MOUNT, type MOUNT followed by the name of the device (for example, DF1:). MOUNT then reads the file DEVS:MOUNTFILE, finds the description for that device, and makes it available for use. The device is not actually initialized, however, until it is first referenced.

Examples:

```
MOUNT prin:
```

mounts an intelligent printer device 'prin:'.

NEWCLI

Format: NEWCLI

Template: NEWCLI "FROM"

Purpose: To create a task associated with a new interactive CLI task.

Specification:

Tripos creates a new CLI. The new task has the same set directory and prompt string as the one where NEWCLI is executed. Each CLI task is independent, allowing separate input, output, and program execution.

You can also use this to create a new CLI on another terminal. The device AUX: refers to a standard alternate terminal. You specify AUX: when creating a new CLI on the device attached to the auxiliary serial port.

If your computer has extra serial ports, then you must to MOUNT them before you can use them.

Examples:

```
NEWCLI
```

creates a new CLI task.

```
NEWCLI AUX:
```

creates a new CLI using the auxiliary serial port (AUX).

Note: Unlike a background task created with the RUN command, a NEWCLI task stays around after you have created it.

See also: ENDCLI, RUN

PATH

Format: PATH [[ADD] <dir1> [<dir2> [...<dir10>]]] [SHOW]

Template: PATH "ADD/S,,,,,,,,,SHOW/S"

Purpose: To alter the directory search list for commands.

Specification:

Initially when you type a command and press RETURN, Tripos searches for that command in your current directory and then in C:. However, you may wish to alter this. The PATH command lets you specify a new list of directories for Tripos to search, and the order in which it is to search them. You can specify up to ten directories (which should be sufficient for most purposes). The order in which you specify the directories dictates the order in which they will be searched (that is, "PATH A B" searches A before B).

Examples:

```
PATH
```

clears the list (that is, it sets the user defined part of the list to nothing).

```
PATH :commands
```

specifies that the directory :commands is to be searched.

```
PATH :commands :tripos/commands :extra/commands
```

specifies that the directory :commands is to be searched first, then :tripos/commands, and then :extra/commands.

```
PATH ADD newcommands
```

adds the directory newcommands to the end of the search list (for example, it would be searched after the directory :extra/commands in the previous example).

```
PATH SHOW
```

displays the current search list on the screen.

PROMPT

Format: PROMPT <prompt>

Template: PROMPT "PROMPT"

Purpose: To change the prompt in the current CLI.

Specification:

If you do not give a parameter, then Tripos resets the prompt to the standard string ("> "). Otherwise, the prompt is set to the string you supply. Tripos also accepts one special character combination (%N). This is demonstrated in the example below.

Examples:

```
PROMPT
```

resets the current prompt to "> ".

```
PROMPT "%N> "
```

resets the current prompt to "n> ", where n is the current task number. Tripos interprets the special character combination %N as the task number. The double quotes (") are not required if there are no spaces in the new prompt string.

PROTECT

Format: PROTECT [FILE] <filename> [FLAGS <status>]

Template: PROTECT "FILE,FLAGS/K"

Purpose: To set a file's protection status.

Specification:

PROTECT takes a file and sets its protection status.

The keyword FLAGS takes four options: read (r), write (w), delete (d), and execute (e). To specify these options you type an r, w, d, or e after the name of the file. If you omit an option, PROTECT assumes that you do not require it. For instance, if you give all the options except d, PROTECT ensures that you cannot delete the file. Read, write, and delete can refer to any kind of file. Tripos only pays attention to the delete (d) flag in the current release. Users and user programs, however, can set and test these flags if they wish.

Examples:

```
PROTECT prog1 FLAGS r
```

sets the protection status of program 1 as read only.

```
PROTECT prog2 rwd
```

sets the protection of program 2 as read/write/delete.

See also: LIST

QUIT

Format: QUIT [<returncode>]

Template: QUIT "RC"

Purpose: To exit from a command sequence with a given error code.

Specification:

QUIT reads through the command file and then stops with an error code. The default error code is zero.

Examples:

```
QUIT
```

exits the current command sequence.

```
FAILAT 30
IF ERROR
QUIT 20
ENDIF
```

If the last command was in error, this terminates the command sequence with error code 20.

For more on command sequences, see the specification for the C command earlier in this chapter.

See also: C, IF, LAB, SKIP

RELABEL

Format: RELABEL [DRIVE] <drive> [NAME] <name>

Template: RELABEL "DRIVE/A,NAME/A"

Purpose: To change the volume name of a disk.

Specification:

RELABEL changes the volume name of a disk to the <name> you specify. Volume names are set initially when you format a disk.

Examples:

```
RELABEL df1: "My other disk"
```

See also: FORMAT

RENAME

Format: RENAME [FROM] <name> [TO|AS] <name>

Template: RENAME "FROM/A,TO=AS/A"

Purpose: To rename a file or directory.

Specification:

RENAME renames the FROM file with the specified TO name. FROM and TO must be filenames on the same disk. The FROM name may refer to a file or to a directory. If the filename refers to a directory, RENAME leaves the contents of the directory unchanged (that is, the directories and files within that directory keep the same contents and names).

Only the name of the directory is changed when you use RENAME. If you rename a directory, or if you use RENAME to give a file another directory name (for example, rename :bill/letter as :mary/letter), Tripos changes the position of the directory, or file, in the filing system hierarchy. Using RENAME is like changing the title of a file and then moving it to another section or drawer in the filing cabinet. Some other systems describe the action as 'moving' a file or directory.

Note: If you already have a file with exactly the same name as the TO file, RENAME won't work. This should stop you from overwriting your files by accident.

Examples:

```
RENAME work/prog1 AS :arthur/example
```

renames the file 'work/prog1' as the file 'arthur/example'. The root directory must contain 'arthur' but not 'arthur/example' for this command to work.

RUN

Format: RUN <command>

Template: RUN command+
command .....

Purpose: To execute commands as background tasks.

Specification:

RUN creates a non-interactive Command Line Interpreter (CLI) task and gives it the rest of the command line as input. The background CLI executes the commands and then deletes itself.

The new CLI has the same set directories and command stack size as the CLI where you called RUN.

To separate commands, type a plus sign (+) and press RETURN. RUN interprets the next line after a + (RETURN) as a continuation of the same command line. Thus, you can make up a single command line of several physical lines that each end with a plus sign.

RUN displays the task number of the newly created task.

Examples:

```
RUN COPY :t/0 PAR:+
DELETE :t/0+
ECHO "Printing finished"
```

copies the file to the printer, deletes it, and displays the message.

```
RUN C comseq
```

executes in the background all the commands in the file 'comseq'.

SEARCH

Format: SEARCH [FROM] <name>|<pat> [SEARCH] <string> [ALL]

Template: SEARCH "FROM,SEARCH/A,ALL/S"

Purpose: To look for a text string you specify in all the files in a directory.

Specification:

SEARCH looks through all the files in the specified directory, and any files in subdirectories if you specify ALL. SEARCH displays any line that contains the text you specified as SEARCH. It also displays the name of the file currently being searched.

You can also replace the directory FROM with a pattern. (See the command LIST for a full description of patterns.) If you use a pattern, SEARCH only looks through files that match the specified pattern. The name may also contain directories specified as a pattern.

Tripos looks for either upper or lower case of the search string. Note that you must place quotation marks around any text containing a space.

As usual, to abandon the command, press CTRL-C, the attention flag. To abandon the search of the current file and continue on to the next file, if any, press CTRL-D.

Examples:

```
SEARCH SEARCH vflag
```

searches through the files in the current directory looking for the text 'vflag'.

```
SEARCH df0: "Happy day" ALL
```

looks for files containing the text 'Happy day' on the entire disk 'df0:'.

```
SEARCH test-#? vflag
```

looks for the text 'vflag' in all files in the current directory starting with 'test-'.

SET-SERIAL

Format: SET-SERIAL <name> [[BAUD] [<rate>]]
[[PARITY] [EVEN|ODD|NONE]]
[[DATABITS] [<dbit>]][[STOPBITS][<sbit>]]

Template: SET-SERIAL "NAME/A,BAUD,PARITY,
DATABITS,STOPBITS"

Purpose: To alter the serial line speeds and data formats.

Specification:

You can use SET-SERIAL to alter the baud rate of any of the serial lines and to specify the parity checking, the number of data bits, and the number of stop bits. SET-SERIAL can also return the current settings of all of these, if you require.

The first argument, <name>, is the name of the serial port you wish to alter; only one port can be altered at a time. The possible options are any serial line device; for example, CON (for CONsole) and AUX (for AUXilliary port).

The other parameters are both optional and positional.

<rate> is the baud rate; it can be any allowable speed on the device (for example, 19200, 9600, 4800, 2400, 1200, 600, or 300), depending on the rate you require.

PARITY can be set to EVEN, ODD, or NONE.

<dbit> is the number of data bits. You can specify the data bits as 5, 6, 7, or 8.

<sbit> is the number of stop bits. You can specify 1, 1.5, or 2 bits.

The line characteristics must be the same as the other device connected to the serial line in question. If you omit any argument altogether, the parameter is left as its current setting. The default settings are as follows:

```
BAUD 9600 PARITY NONE DATABITS 8 STOPBITS 1
```

Current settings are not individually displayable. You can obtain all the current settings for a particular device by specifying its name after the command. For example, to display all the current settings for the console device, type

```
SET-SERIAL con
```

Examples:

```
SET-SERIAL con BAUD 9600 PARITY even
```

sets the console serial line to 9600 baud and even parity. The data and stop bits are unchanged and remain at their current setting.

SKIP

Format: SKIP <label>

Template: SKIP "LABEL"

Purpose: To perform a jump in a command sequence.

Specification:

SKIP can be used only within a C command file. You use SKIP in conjunction with LAB. (See LAB for details.) SKIP reads through the command file looking for a label you defined with LAB, without executing any commands.

You can use SKIP either with or without a label; without one, it finds the next unnamed LAB command. With one, it attempts to find a LAB defining a label, as specified. LAB must be the first item on a line of the file. If SKIP does not find the label you specified, the sequence terminates and Tripos displays the following message:

```
label "<label>" not found by Skip
```

SKIP only jumps forwards in the command sequence.

Examples:

```
SKIP
```

skips to the next LAB command without a name following it.

```
IF ERROR

SKIP errlab

ENDIF
```

If the last command stopped with a return code >= 20, this searches for the label 'errlab' later in the command file.

```
FAILAT 100
ASSEM text
IF ERROR
SKIP ERROR
ENDIF
LINK
SKIP DONE
LAB ERROR
ECHO "Error doing Assem"
LAB DONE
ECHO "Next command please"
```

See also: C, LAB, IF, FAILAT, QUIT

SORT

Format: SORT [FROM] <name> [[TO] <name>] [COLSTART <n>]

Template: SORT "FROM/A,TO/A,COLSTART/K"

Purpose: To sort simple files.

Specification:

This command is a very simple sort package. You can use SORT to sort files although it isn't fast for large files, and it cannot sort files that don't fit into memory.

You specify the source as FROM, and the sorted result goes to the file TO. SORT assumes that FROM is a normal text file where each line is separated with a carriage return. Each line in the file is sorted into increasing alphabetic order without distinguishing between upper and lower cases.

To alter this in a very limited way, use the COLSTART keyword to specify the first column where the comparison is to take place. SORT then compares the characters on the line from the specified starting position to the end; if the lines still match after this, then the remaining columns from the first to just before the column specified as COLSTART are included in the comparison.

Note: The initial stack size (that is, 4000 bytes) is only suitable for small files of less than 200 lines or so. If you want to sort larger files, you must use the STACK command to increase the stack size; how much you should increase the size is part skill and part guesswork.

WARNING: The computer will crash if STACK is too small. If you are not sure, it is better to overestimate the amount you need.

Examples:

```
SORT text TO sorted-text
```

sorts each line of information in 'text' alphabetically and places the result in 'sorted-text'.

```
SORT index TO sorted-index COLSTART 4
```

sorts the file 'index', where each record contains the page number in the first three columns and the index entry on the rest of the line, and puts the output in 'sorted-index' sorted by the index entry, and matching index entries sorted by page number.

See also: ><, STACK

STACK

Format: STACK [<n>]

Template: STACK "SIZE"

Purpose: To display or set the stack size for commands.

Specification:

When you run a program, it uses a certain amount of stack space. In most cases, the initial stack size, 4000 bytes, is sufficient, but you can alter it using the STACK command. To do this, you type STACK followed by the new stack value. You specify the value of the stack size in bytes. STACK alone displays the currently set stack size.

Usually, only SORT requires an increased stack size. Recursive commands such as DIR need an increased stack if you use them on a directory structure more than about six levels deep.

WARNING: The only indication that you have run out of stack is that the computer crashes! If you are not sure, overestimate.

Examples:

STACK

displays the current stack size.

STACK 8000

sets the stack to 8000 bytes.

See also: RUN, SORT

STATUS

Format: STATUS [<task>] [FULL] [TCB] [SEGS] [CLI|ALL]

Template: STATUS "TASK,FULL/S,TCB/S,SEGS/S, CLI=ALL/S"

Purpose: To display information about the currently existing CLI tasks.

Specification:

STATUS alone lists the numbers of tasks and the program running in each.

TASK specifies a task number and only gives information about that task. Otherwise, information is displayed about all tasks.

FULL = SEGS + TCB + CLI

SEGS displays the names of the sections on the segment list of each task.

TCB displays information about the priority, stacksize, and global vector size of each task.

For further details on stack and global vector size, see the *Tripos Technical Reference Manual*.

CLI identifies Command Line Interpreter tasks and displays the section name(s) of the currently loaded command (if any).

Examples:

```
STATUS 4 FULL
```

displays full information about task 4.

TYPE

Format: TYPE [FROM] <name> [[TO] <name>] [OPT N|H]

Template: TYPE "FROM/A,TO,OPT/K"

Purpose: To type a text file or to type a file out as hexadecimal numbers.

Specification:

TO indicates the output file that you specify; if you omit this, output is to the current output stream, which means, in most cases, that the output goes to the screen.

To interrupt output, press CTRL-C. To suspend output, type any character. To resume output, press RETURN or CTRL-X.

OPT specifies an option to TYPE. The first option to TYPE is 'n', which includes line numbers in the output.

The second option you can give TYPE is h. Use the h option to write out each word of the FROM file as a hex number, with the character representation in a column down the right-hand side.

Examples:

```
TYPE work/prog
```

displays the file 'work/prog'.

```
TYPE work/prog OPT n
```

displays the file 'work/prog' with line numbers.

```
TYPE obj/prog OPT h
```

displays the code stored in 'obj/prog' in hexadecimal.

VDU

Format: VDU [<name>]

Template: VDU "NAME"

Purpose: To identify the make of terminal in use.

Specification:

You follow the VDU command with the name of a VDU that you are using. Until you have given the VDU command, a number of commands (for example, ED) won't work. The VDU command opens the file DEVS:VDU and locates the name you specifed. If you wish to find out which VDUs are currently supported, you can look at the file DEVS:VDU with the editor. However, you may have to make certain alterations to this file to support new makes of terminal. (See Chapter 4, "Installation", in the *Tripos Technical Reference Manual* for further details.)

There is a limit on the size of a vdu specification and this limit cannot be altered. If it is exceeded or any syntax error is found, then a suitable message is given and the vdu specification is not altered.

The VDU handler is installed for the current task and for all tasks created from that task by NEWCLI. Normally, you give the VDU command once when you startup the system, and so you may find it useful to place it in the file S:STARTUP-SEQUENCE. Once you have done this, Tripos executes it automatically on starting up.

If you omit the vdu name, the name of the current vdu type appears on the screen.

Examples:

```
VDU TVI
```

loads a VDU driver for the Televideo 950.

WAIT

Format: WAIT <n> [SEC|SECS] [MIN|MINS] [UNTIL <time>]

Template: WAIT ",SEC=SECS/S,MIN=MINS/S,UNTIL/K"

Purpose: To wait for the specified time.

Specification:

You can use WAIT in command sequences or after RUN to wait for a certain period, or to wait until a certain time of day. Unless you specify otherwise, the waiting time is one second.

The parameter should be a number, specifying the number of seconds (or minutes, if MINS is given) to wait.

Use the keyword UNTIL to wait until a specific time of day, given in the format HH:MM.

Examples:

```
WAIT
```

waits 1 second.

```
WAIT 10 MINS
```

waits 10 minutes.

```
WAIT UNTIL 21:15
```

waits until quarter past nine at night.

WHY

Format: WHY

Template: WHY

Purpose: To explain why the previous command failed.

Specification:

Usually when a command fails the screen displays a brief message that something went wrong. This typically includes the name of the file (if that was the problem), but does not go into any more detail. For example, the command

```
COPY fred TO *
```

might fail and display the message

```
Can't open fred
```

This could happen for a number of reasons - for example, 'fred' might already be a directory, or there might not be enough space on the disk to open the file, or it might be a read-only disk. COPY makes no distinction between these cases, because usually the user knows what is wrong. However, immediately after your command fails, type WHY and press RETURN to display a much fuller message, describing in detail what went wrong.

Examples:

```
TYPE DF0:
can't open DF0:
WHY
Last command failed because object not of required type
```

WHY hints at why your command failed: you can't type a device.

See also: FAULT

Quick Reference Card

File Utilities

;	comment character, ignore the rest of the line.
< >	direct command input and output respectively.
COPY	copies one file to another or copies all the files from one directory to another.
DELETE	deletes up to 10 files or directories.
DIR	shows filenames in a directory.
ED	enters a screen editor for text files.
EDIT	enters a line by line editor.
FILENOTE	attaches a note with a maximum of 80 characters to a specified file.
JOIN	concatenates up to 15 files to form a new file.
LIST	examines and displays detailed information about a file or directory.
MAKEDIR	creates a directory with a specified name.
PROTECT	sets a file's protection status.
RENAME	renames a file or directory.

SEARCH	looks for a specified text string in all the files of a directory.
SORT	sorts simple files.
TYPE	types a file to the screen that you can optionally specify as text or hex.

CLI Control

BREAK	sets attention flags in a given task.
CD	sets a current directory and/or drive.
CONSOLE	sets the console characteristics of the console handler used by the current CLI.
ENDCLI	ends an interactive CLI task.
NEWCLI	creates a new interactive CLI task.
PROMPT	changes the prompt in the current CLI.
RUN	executes commands as background tasks.
STACK	displays or sets the stack size for commands.
STATUS	displays information about the CLI tasks currently in existence.
VDU	identifies the make of terminal in use.
WHY	explains why a previous command failed.

Command Sequence Control

C	executes a file of commands.
ECHO	displays the message specified in a command argument.
FAILAT	fails a command sequence if a program returns an error code greater than or equal to this number.
IF	tests specified actions within a command sequence.
LAB	defines a label (see SKIP).
QUIT	exits from a command sequence with a given error code.
SKIP	jumps forward to LAB in a command sequence (see LAB).
WAIT	waits for, or until, a specified time.

System and Storage Management

ASSIGN	assigns a logical device name to a filing system directory.
DATE	displays or sets the system date and time.

DISKCOPY	copies the contents of one entire floppy disk to another.
DISKDOCTOR	restores a corrupt disk.
FAULT	displays messages corresponding to supplied fault or error codes.
FORMAT	formats and initializes a new floppy disk.
INFO	gives information about the filing system.
INSTALL	makes a formatted disk bootable.
MOUNT	mounts a new device.
PATH	alters the directory search list for commands.
RELABEL	changes the volume name of a disk.

Programming Tools

ALINK	links sections of code into a file for execution (see JOIN).
ASSEM	assembles M68000 language.

Chapter 2: ED - The Screen Editor

This chapter describes how to use the screen editor ED. You can use this program to alter or create text files.

Table of Contents

2.1 Introducing ED

You can use the editor ED to create a new file or to alter an existing one. You display text on the screen, and you can scroll it vertically or horizontally, as required.

ED accepts the following template:

```
ED "FROM/A,SIZE/K"
```

For example, to call ED, you type

```
ED fred
```

ED makes an attempt to open the file you have specified as 'fred' (that is, the FROM file), and if this succeeds, then ED reads the file into memory and displays the first few lines on the screen. Otherwise, ED provides a blank screen, ready for the addition of new information. To alter the text buffer that ED uses to hold the file, you specify a suitable value after the SIZE keyword, for example,

```
ED fred SIZE 45000
```

The initial size is based on the size of the file you edit, with a minimum of 20,000 words.

Note: You cannot edit every kind of file with ED. For example, ED does not accept source files containing binary code. To edit files such as these, you should use the editor EDIT.

WARNING: ED always appends a linefeed even if the file does not end with one.

When ED is running, the bottom line of the screen is a message area and command line. Error messages appear here and remain until you give another ED command.

ED commands fall into two categories:

- immediate commands
- extended commands.

You use immediate commands in **immediate mode**; you use extended commands in **extended mode**. ED is already in immediate mode when you start editing. To enter extended mode, you press the ESC key. Then, after ED has executed the command line, it returns automatically to immediate mode.

In immediate mode, ED executes commands right away. You specify an immediate command with a single key or control key combination. To indicate a control key combination, you press and hold down the CTRL key while you type the given letter, so that CTRL-M, for example, means hold down CTRL while you type M.

In extended mode, anything you type appears on the command line. ED does not execute commands until you finish the command line. You may type a number of extended commands on a single command line. You may also group any commands together and even get ED to repeat them automatically. Most immediate commands have a matching extended version.

ED attempts to keep the screen up to date. However, if you enter a further command while it is attempting to redraw the display, ED executes the command at once and updates the display when there is time. The current line is always displayed first and is always up to date.

2.2 Immediate Commands

This section describes the type of commands that ED executes immediately. Immediate commands deal with the following:

- cursor control
- text insertion
- text deletion
- text scrolling
- repetition of commands

2.2.1 Cursor Control

To move the cursor one position in any direction, you press the appropriate cursor control key. If you do not have cursor control keys (keys with arrows on them), then you can use the following control combinations: CTRL-H to move the cursor right, CTRL-J to move the cursor down, CTRL-K to move the cursor up, and CTRL-X to move the cursor right.

If the cursor is on the right hand edge of the screen, ED scrolls the text to the left to make the rest of the text visible. ED scrolls vertically a line at a time and horizontally ten characters at a time. You cannot move the cursor off the top or bottom of the file, or off the left hand edge of the text.

HOME (or CTRL-], that is, CTRL and the square closing bracket ']') takes the cursor to the right hand edge of the current line unless the cursor is already there. When the cursor is already at the right hand edge, HOME (or CTRL-]) moves it back to the left hand edge of the line. The text is scrolled horizontally, if required. In a similar fashion, CTRL-E places the cursor at the start of the first line on the screen unless the cursor is already there. If the cursor is already there, CTRL-E places it at the end of the last line on the screen.

CTRL-T takes the cursor to the start of the next word. CTRL-R takes the cursor to the space following the previous word. In these two cases, the text is scrolled vertically or horizontally, as required.

The TAB key (also CTRL-I) moves the cursor to the next tab position, which is a multiple of the tab setting (initially 3). It does NOT insert TAB characters in the file.

2.2.2 Inserting Text

While in immediate mode, ED is also in INSERT mode so any ordinary characters you type will be inserted at the current cursor position. ED has no type-over mode. To replace a word or line, you must delete the desired contents and insert new information in its place. Any letter that you type in immediate mode appears at the **current cursor position** unless the line is too long (there is a maximum of 255 characters in a line). If you try to make a line longer than the maximum limit, ED refuses to add another character and displays the following message:

```
Line too long
```

However, on shorter lines, ED moves any characters to the right of the cursor to make room for the new text. If the line exceeds the size of the screen, the left hand end of the line disappears from view. Then, ED redisplays the end of the line by scrolling the text horizontally. If you move the cursor beyond the end of the line, for example, with the TAB or cursor control keys, ED inserts spaces between the end of the line and any new character you insert.

To split the current line at the cursor and generate a new line, press RETURN. If the cursor is at the end of a line, ED creates a new blank line after the current one. Alternatively, you press CTRL-A (or the INS LINE key, if there one is on your make of terminal) to generate a blank line after the current one, with no split of the current line taking place. In either case, the cursor appears on the new line at the position indicated by the left margin (initially, column one).

To ensure that ED gives a carriage return automatically at a certain position on the screen, you can set up a right margin. Once you have done this, whenever you type a line that exceeds that margin, ED ends the line before the last word and moves the word and the cursor down onto a new line. This is called 'word wrap.' (Note that if you have a line with no spaces, ED won't know where to break the 'word' and the automatic

margin cannot work properly.) In detail, if you type a character and the cursor is at the end of the line and at the right margin position, then ED automatically generates a new line. Unless the character you typed was a space, ED moves down the half completed word at the end of the line to the newly generated line. However, if you insert some text when the cursor is NOT at the end of a line (that is, with text already to the right of the cursor), then setting a right margin does not work. Initially, the right margin is set up at column 79. You can turn off, or 'disable,' the right margin with the EX command. (For further details on setting margins, see Section 2.2.1 "Program Control").

If you type some text in the wrong case (for example, in lower case instead of upper case), you can correct it with CTRL-F. To do this, you move the cursor to point at the letter you want to change and then press CTRL-F. If the letter is in lower case, CTRL-F flips the letter into upper case. On the other hand, if the letter is in upper case, CTRL-F flips it into lower case. However, if the cursor points at something that is not a letter (for example, a space or symbol), CTRL-F does nothing to it.

CTRL-F not only flips letter cases but it also moves the cursor one place to the right (and it moves the cursor even if there is no case to flip). So that, after you have changed the case of a letter with CTRL-F, the cursor moves right to point at the next character. If the next character is a letter, you can press CTRL-F again to change its case; you can then repeat the command until you have changed all the letters on the line. (Note that if you continue to press CTRL-F after the the last letter on the line, the cursor keeps moving right even though there is nothing left to change.) For example, if you had the line

```
The Walrus and the Carpenter were walking hand in hand
```

and you kept CTRL-F pressed down, the line would become

```
tHE wALRUS AND THE cARPENTER WERE WALKING HAND IN HAND
```

On the other hand, the following line:

```
IF <file> <= x
```

becomes

```
if <FILE> <= X
```

where the letters change case and the symbols remain the same.

2.2.3 Deleting Text

The BACKSPACE key deletes the character to the left of the cursor and moves the cursor one position left unless it is at the beginning of a line. ED scrolls the text, if required. CTRL-N (also DEL, or DEL CHAR on some terminals) deletes the character at the current cursor position without moving the cursor. As with any deletion, characters remaining on the line shift back, and text that was invisible beyond the right hand edge of the screen becomes visible.

The action of CTRL-O depends on the character at the cursor. If this character is a space, then CTRL-O (also INS CHAR on some terminals) deletes all spaces up to the next non-space character on the line. Otherwise, it deletes characters from the cursor, and moves text left, until a space occurs.

CTRL-Y (also LINE ERASE, or DEOL on some terminals) deletes all characters from the cursor to the end of the line.

CTRL-B (also DEL LINE on some terminals) deletes the entire current line. You may use extended commands to delete blocks of text.

2.2.4 Scrolling

Besides vertically scrolling one line line at a time by moving the cursor to the edge of the screen, you can vertically scroll the text 12 lines at a time with the control keys CTRL-U and CTRL-D.

CTRL-D moves the cursor to previous lines, while scrolling the text down; CTRL-U scrolls the text up and moves the cursor to lines further on in the file.

CTRL-V refreshes the entire screen, which is useful if another program besides the editor alters the screen.

2.2.5 Repeating Commands

The editor remembers any extended command line you type. To execute this set of extended commands again at any time, press CTRL-G. In this way, you can set up a search command as an extended command. If the first occurrence of a string is not the one you need, press CTRL-G to repeat the search. You can set up and execute complex sets of editing commands many times.

Note: When you give an extended command as a command group with a repetition count, ED repeats the commands in the group that number of times each time you press CTRL-G. See the section "Extended Commands" for more details on extended commands.

2.3 Extended Commands

This section describes the commands available to you in extended mode. These commands cover

- program control
- block control
- movement
- searching text
- exchanging text
- altering text
- inserting text

To enter extended command mode, press the ESC or ESCAPE key. Some makes of terminal, however, send the ESCAPE character as a prefix to function keys. If your terminal is one of these, you must make it accept another key instead. This can be done on installation (see the *Tripos Technical Reference Manual* for further details). You can then use that key wherever you would use ESC. Once you have pressed ESC, ESCAPE, or the key you have set up to be equivalent to ESCAPE, all subsequent input appears on the command line at the bottom of the

screen. You can correct mistakes with BACKSPACE in the normal way. To terminate the command line, press either ESC or RETURN. If you press ESC, the editor remains in extended mode after executing the command line. On the other hand, if you press RETURN, it reverts to immediate mode. To leave the command line empty, just press RETURN after pressing ESC to go back to immediate mode. In this case, ED returns to immediate command mode.

Extended commands consist of one or two letters, with upper and lower case considered the same. You can give multiple commands on the same command line by separating them with a semicolon. Commands are sometimes followed by an argument, such as a number or a string. A string is a sequence of letters introduced and terminated by a delimiter, which is any character except letters, numbers, space, semicolon, or brackets. Thus, valid strings might be

```
/happy/   !23 feet!  :Hello!: "1/2"
```

Most immediate commands have a corresponding extended version. See the Table of Extended Commands at the end of this chapter for a complete list.

2.2.1 Program Control

This section provides a specification of the program control commands X (eXit), Q (Quit), SA (SAve), U (Undo), SH (SHow), ST (Set Tab), SL and SR (Set Left and Set Right), and EX (EXtend).

To instruct the editor to exit, you use the command X. After you have given the exit command, ED writes out the text it is holding in memory to the output, or destination file and then terminates. If you look at this file, you can see that all the changes you made are there.

ED also writes a temporary backup to :T/ED-BACKUP. This backup file remains until you exit from ED again, at which time, ED overwrites the file with a new backup.

To get out of the editor without keeping any changes, you use the Q command. When you use Q, ED terminates immediately without writing to the buffer and discards any changes you have made. Because of this, if you have altered the contents of the file, ED asks you to confirm that you really want to quit.

A further command lets you to take a 'snapshot' copy of the file without coming out of ED. This is the SA command. SA saves the text to a named file or, in the absence of a named file, to the current file. For example,

```
SA !:doc/savedtext!
```

or

```
SA
```

SA is particularly useful in geographical areas subject to power failure or surge.

Hint: SA followed by Q is equivalent to the X command.

If you make any alterations between the SA and the Q commands, the following message appears:

```
Edits will be lost - type Y to confirm:
```

If you have made no alterations, ED quits immediately with the contents of your source file unchanged. SA is also useful because it allows you to specify a filename other than the current one. It is therefore possible to make copies at different stages and place them in different files or directories.

To undo the last change, you use the U command. The editor makes a copy of the line the cursor is on, and then it modifies this copy whenever you add or delete characters. ED puts the changed copy back into the file when you move the cursor off the current line (either by cursor control, or by deleting or inserting a line). ED also replaces the copy when it performs any scrolling either vertically or horizontally. The U command discards the changed copy and uses the old version of the current line instead.

WARNING: ED does not undo a line deletion. Once you have moved from the current line, the U command cannot fix the mess you have got yourself into.

To show the current state of the editor, you use the SH command. The screen displays information such as the value of tab stops, current margins, block marks, and the name of the file being edited.

Tabs are initially set at every three columns. To change the current setting of tabs, you use the ST command followed by a number n, which sets tabs at every n columns.

To set the left margin and right margin, you use the SL and SR commands, again followed by a number indicating the column position. The left margin should not be set beyond the width of the screen.

To extend margins, you use the EX command. Once you have given EX, ED takes no account of the right margin on the current line. Once you move the cursor from the current line, ED turns the margins on again.

2.3.2 Block Control

To move, insert, or delete text, you use the block control commands described in this section.

You can identify a block of text with the BS (Block Start) and BE (Block End) commands. To do this, move the cursor to anywhere on the first line you that you want in the block and give the BS command. Then, move the cursor to the last line that you want in the block, using the cursor control commands or a search command, and give the BE command to mark the end of the block.

Note: Once you have defined a block with BS and BE, if you make ANY change to the text, the start and end of the block become undefined once more. The only exception to this is if you use IB (Insert Block).

To identify one line as the current block, move to the line you want, press ESC, and type

```
BS;BE
```

The current line then becomes the current block.

Note: You cannot start or finish a block in the middle of a line. To do this, you must first split the line by pressing RETURN.

Once you have identified a block, you can move a copy of it into another part of the file with the IB (Insert Block) command. When you give the IB command, ED inserts a copy of the block immediately after the current line. You can insert more than one copy of the block, as it remains defined until you change the text, or delete the block.

To delete a block, you use the DB (Delete Block) command. DB deletes the block of text you defined with the BS and BE commands. However, when you have deleted the block, the block start and end values become undefined. This means that you cannot delete a block and then insert a copy of it (DB followed by IB); however, you can insert a copy of the block and then delete the block (IB followed by DB).

You can also use block marks to remember a place in a file. The SB (Show Block) command resets the screen window on the file so that the first line in the block is at the top of the screen.

To write a block to another file, you use the WB command (Write Block). This command takes a string that represents a file name. For example,

```
WB !:doc/example!
```

writes the contents of the block to the file 'example' in the directory ':doc'. (Remember: if you use the filename-divider slash (/) to separate directories and files, you should not use slash as a delimiter). ED then creates a file with the name that you specified, possibly destroying a previous file with that name and finally writes the buffer to it.

To insert a file into the current file, you use the IF command (Insert File). ED reads into memory the file with the name you gave as the argument string to IF, at the point immediately following the current line. For example,

```
IF !:doc/example!
```

inserts the file :doc/example into the current file beginning immediately after the current line.

2.3.3 Moving the Current Cursor Position

The command T moves the cursor to the top of the file, so that the first line in the file is the first line on the screen. The B command moves the cursor to the bottom of the file, so that the last line in the file is the bottom line on the screen.

The commands N and P move the cursor to the start of the next line and previous line, respectively. The commands CL and CR move the cursor one place to the left or one place to the right, while CE places the cursor at the end of the current line, and CS places it at the start.

The command M moves the cursor to a specific line. To move, you type M followed by the line number of the line you want as the new current line. For example,

```
M 503
```

moves the cursor to the five hundred and third line in the file. The M command is a quick way of reaching a known position in your file. You can, for instance, move to the correct line in your file by giving a repeat count to the N command, but it is much slower.

2.3.4 Searching and Exchanging

Alternatively you can move the screen window to a particular context with the command F (Find) followed by a string that represents the text to be located. The search starts at one place beyond the current cursor position and continues forwards through the file. If the string is found, the cursor appears at the start of the located string. The string must be in quotes (or other delimiters '/', '.', '!', and so on). In order for a match to occur the strings must be of the same case, unless the UC command is used (see below).

To search backwards through the text, you use the command BF (Backwards Find) in the same way as F. BF finds the last occurrence of the string before the current cursor position. (That is, BF looks for the string to the left of the cursor and then through all the lines back to the beginning of the file.) To find the earliest occurrence, you use T (Top-of-file) followed by F. To find the last occurrence, you use B (Bottom-of-file) followed by BF.

The E (Exchange) command takes two strings separated with delimiter characters and exchanges the first string for the last. So, for example,

```
E /wombat/zebra/
```

would change the next occurrence of the text 'wombat' to 'zebra'. The editor starts searching for the first string at the current cursor position and continues through the file. After the exchange is completed, the cursor moves to the end of the exchanged text.

You can specify empty strings by typing two delimiters with nothing between them. If the first, or 'search', string is empty, the editor inserts the second string at the current cursor position. If the second string is empty, the next occurrence of the search string is exchanged for nothing (that is, the search string is deleted).

Note: ED ignores margin settings while you are exchanging text.

The EQ command (Exchange and Query) is a variant on the E command. When you use EQ, ED asks you whether you want the exchange to take

place. This is useful when you want the exhange to take place in some circumstances, but not in others. For example, after typing

```
EQ /wombat/zebra/
```

the following message

Exchange?

appears on the command line. If you respond with an N, then the cursor moves past the search string; otherwise, if you type Y, the change takes place as normal. You usually only give EQ in repeated groups.

The search and exchange commands usually make a distinction between upper and lower case while making the search. To tell all subsequent searches not to make any distinction between upper and lower case, you use the UC command. Once you have given UC, the search string 'wombat' matches 'Wombat', 'WOMBAT', 'WoMbAt' and so on. To have ED distinguish between upper and lower case again, you use LC.

2.3.5 Altering Text

You cannot use the E command to insert a new line into the text. You use the I and A commands instead. Follow the I command (Insert before) with a string that you want to make into a new line. ED inserts this new line before the current line. For example,

```
I /Insert this BEFORE the current line/
```

inserts the string 'Insert this BEFORE the current line' as a new, separate line Before the line containing the cursor. You use the A command (insert After) in the same way except that ED inserts the new line after the current line. That is,

```
A /Insert this AFTER the current line/
```

inserts the string 'Insert this AFTER the current line' as a new line After the line containing the cursor.

To split the current line at the cursor position, you use the S command. S in extended mode is just like pressing RETURN in immediate mode (see Section 2.2.2 for further details on splitting lines).

The J command joins the next line onto the end of the current one.

The D command deletes the current line in the same way as CTRL-B in immediate mode. The DC command deletes the character above the cursor in the same way as CTRL-N.

2.3.6 Repeating Commands

To repeat any command a certain number of times, precede it with the desired number. For example,

```
4 E /slithy/brillig/
```

changes the next four occurrences of 'slithy' to 'brillig'. ED verifies the screen after each command. You use the RP (Repeat) command to repeat a command until ED returns an error, such as reaching the end of the file. For example,

```
T; RP E /slithy/brillig/
```

changes all occurrences of 'slithy' to 'brillig'. Notice that you need the T command to ensure that all occurrences of 'slithy' are changed, otherwise only those after the current position are changed.

To execute command groups repeatedly, you can group the commands together in parentheses. You can also nest command groups within command groups. For example,

```
RP ( F /bandersnatch/; 3 A/ / )
```

inserts three blank lines (copies of the null string) after every line containing 'bandersnatch'. Notice that this command line only works from the cursor to the end of the file. To apply the command to every line in the file, you should first move to the top of the file.

Note that some commands are possible, but silly. For example,

```
RP SR 60
```

sets the right margin to 60 *ad infinitum*. However, to interrupt any sequence of extended commands, and particularly repeated ones, you type any character while the commands are taking place. If an error occurs, ED abandons the command sequence.

2.3.7 Executing Tripos Commands in ED

You may wish to execute a Tripos command without having to exit ED first. To switch to another task (process), created with NEWCLI, press CTRL-P as usual. You will then find yourself switched to a new task that is waiting for input. To return to ED, keep pressing CTRL-P until you find yourself in ED. (CTRL-P cycles through all the tasks that are currently waiting for input.)

If you only wish to execute one command before returning to ED, you may find it simpler to use DO. For example, to halt ED and list the files in the root directory, type

```
DO/LIST/
```

Then press RETURN to return to ED again.

Quick Reference Card

Special Key Mappings

Command	Action
BACKSPACE	Delete character to left of cursor
ESC	Enter extended command mode
RETURN	Split line at cursor and create a new line
TAB	Move cursor right to next tab position (does NOT insert a TAB character)
<up-arrow>	Move cursor up
<down-arrow>	Move cursor down
<left-arrow>	Move cursor left
<right-arrow>	Move cursor right

Immediate Commands

Command	Action
CTRL-A	Insert line
CTRL-B	Delete line
CTRL-D	Scroll text down
CTRL-E	Move to top or bottom of screen
CTRL-F	Flip case
CTRL-G	Repeat last extended command line
CTRL-H	Delete character left of cursor
CTRL-I	Move cursor right to next tab position
CTRL-M	Return
CTRL-N	Delete character at cursor
CTRL-O	Delete word or spaces
CTRL-R	Cursor to end of previous word

Command	Action
CTRL-T	Cursor to start of next word
CTRL-U	Scroll text up
CTRL-V	Verify screen
CTRL-Y	Delete to end of line
CTRL-[	Escape (enter extended mode)
CTRL-]	Cursor to end or start of line

Extended Commands

This is a full list of extended commands including those that are merely extended versions of immediate commands. In the list, /s/ indicates a string, /s/t/ indicates two exchange strings, and n indicates a number.

Command	Action
A /s/	Insert line after current
B	Move to bottom of file
BE	Block end at cursor
BF /s/	Backwards find
BS	Block start at cursor
CE	Move cursor to end of line
CL	Move cursor one position left
CR	Move cursor one position right
CS	Move cursor to start of line
D	Delete current line
DB	Delete block
DC	Delete character at cursor
DO /s/	Halt ED and execute s
E /s/t/	Exchange s into t
EQ /s/t/	Exchange but query first
EX	Extend right margin
F /s/	Find string s
I /s/	Insert line before current
IB	Insert copy of block

Command	Action
IF /s/	Insert file s
J	Join current line with next
LC	Distinguish between upper and lower case in searches
M n	Move to line number n
N	Move to start of next line
P	Move to start of previous line
Q	Quit without saving text
RP	Repeat until error
S	Split line at cursor
SA	Save text to file
SB	Show block on screen
SH	Show information
SL n	Set left margin
SR n	Set right margin
ST n	Set tab distance
T	Move to top of file
U	Undo changes on current line
UC	Equate U/C and l/c in searches
WB /s/	Write block to file s
X	Exit, writing text into memory

Chapter 3: EDIT - The Line Editor

This chapter describes in detail how to use the line editor EDIT. The first part introduces the reader to the editor. The second part gives a complete specification of EDIT. There is a quick reference card containing all the EDIT commands at the end of the chapter.

Table of Contents

3.1 Introducing EDIT

EDIT is a text editor that processes **sequential files** line by line under the control of **editing commands**. EDIT moves through the input, or **source file**, passing each line (after any possible alterations) to a sequential output file, the **destination file**. An EDIT run, therefore, makes a copy of the source file that contains any changes that you requested with the editing commands.

Although EDIT usually processes the source file in a forward sequential manner, it has the capability to move backwards a limited number of lines. This is possible because EDIT doesn't write the lines that have been passed to the destination file immediately, but holds them instead in an **output queue**. The size of this queue depends on the amount of **memory** available. If you want to hold more information in memory, you can select the EDIT option, OPT, described in the next section, to increase the amount.

You can make more than one pass through the text.

The EDIT commands let you

- change parts of the source
- output parts of the source to other destinations
- insert material from other sources

3.1.1 Calling EDIT

This section describes the format of the **arguments** you can give every time you call the EDIT command. EDIT expects the following arguments:

FROM/A,TO,WITH/K,VER/K,OPT/K

The **command template** described in Chapter 1 of the *Introduction to Tripos* is a method of defining the syntax for each command. Tripos accepts command arguments according to the format described in the command template. For example, some arguments are optional, some

must appear with a keyword, and others do no need keywords because they only appear in a specific position. Arguments with a following /A (like FROM) must appear, but you do not have to type the keyword. Arguments with just a following /K (such as WITH, VER, and OPT) are optional, but you must type the keyword to specify them. Arguments without a following / (TO, for example) are optional. Tripos recognizes arguments without a following slash (/) by their position alone. If you forget the syntax for EDIT, type

```
EDIT ?
```

and Tripos displays the full template on the screen. (For more details on using commands, see Chapter 1 of the *Introduction to Tripos* and Chapter 1 of this manual.)

Using another method of description, the command syntax for EDIT is as follows:

```
[FROM] <file> [[TO] <file>] [WITH <file>] [VER <file>]
[OPT Pn|Wn|PnWn]
```

The argument FROM represents the source file that you want to edit. The argument must appear, but the keyword itself is optional (that is, Tripos accepts the FROM file by its position). It does not require you to type the keyword FROM as well.

The TO file represents the destination file. This is the file where EDIT sends the output including the editing changes. If you omit the TO argument, EDIT uses a temporary file that it renames as the FROM file when editing is complete. If you give the EDIT command STOP, this renaming does not take place, and the original FROM file is untouched.

The WITH keyword represents the file containing the editing commands. If you omit the WITH argument, EDIT reads from the terminal.

The VER keyword represents the file where EDIT sends error messages and line verifications. If you omit the VER argument, EDIT uses the terminal.

You can use the OPT keyword to specify options to EDIT. Valid options are P<n>, which sets the number of previous lines available to the integer <n>, and W<n>, which sets the maximum line length handled to <n> characters. Unless you specify otherwise, Tripos sets the options P40W120.

You can use OPT to increase, or decrease, the size of available memory. EDIT uses P*W (that is, the number of previous lines multiplied by the line width) to determine the available memory. To change the memory size, adjust the P and W numbers. P50 allocates more memory than usual; P30 allocates less memory than usual.

Here are some examples of how you can call EDIT:

```
EDIT programl TO programl_new WITH edit_commands

EDIT programl OPT P50W240

EDIT programl VER ver_file
```

Note: Unlike ED, you cannot use EDIT to create a new file. If you attempt to create a new file, Tripos returns an error because it cannot find the new file in the current directory.

3.1.2 Using EDIT Commands

This section introduces some of the basic EDIT commands omitting many of the advanced features. A complete description of the command syntax and of all commands appears in the Section 3.2, "A Complete Specification of EDIT."

3.1.2.1 The Current Line

As EDIT reads lines from the source and writes them to the destination, the line that it has 'in its hand' at any time is called the current line. EDIT makes all the textual changes to the current line. EDIT always inserts new lines before the current line. When you first enter EDIT, the current line is the first line of the source.

3.1.2.2 Line Numbers

EDIT assigns each line in the source a unique line number. This line number is not part of the information stored in the file, but EDIT computes it by counting the lines as they are read. When you're using EDIT, you can refer to a specific line by using its line number. A line that has been read retains its original line number all the time it is in main memory, even when you delete lines before or after it, or insert some extra lines. The line numbers remain unchanged until you rewind the file, or until you renumber the lines with the = command. EDIT assigns the line numbers each time you enter the file. The line numbers, therefore, may not be the same when you re-enter.

3.1.2.3 Selecting a Current Line

To select a current line in EDIT, you can use one of three methods:

- counting lines
- specifying the line number
- specifying the context

These three methods are described below.

By Line Counting

The N and P commands allow you to move to the next or previous lines. If you give a number before the N or P command, you can move that number of lines forward or backward. To move forward to the next line, type

```
N
```

For any EDIT command, you can type either upper or lower case letters.

To move four lines forward, type

```
4N
```

to make the fourth line from the current line your new current line.

To move back to a line above the current line, type

```
P
```

The P command also takes a number. For example, type

```
4P
```

This makes the fourth line above the current line your new current line. It is only possible to go back to previous lines that EDIT has not yet written to the output. EDIT usually lets you go back 40 lines. To be able to move back more than this, you specify more previous lines with the P option when you enter EDIT (see Section 3.1.1 earlier in this chapter for further details on the P option).

By Moving to a Specific Line Number

The M command allows you to select a new current line by specifying its line number. You type the M command and the desired line number. For example, the command M45 tells EDIT to Move to line 45. If you are beyond line 45, this command moves back to it provided it is still in main memory.

You can combine the specific line number and line counting commands. For example,

```
M12; 3N
```

To separate consecutive commands on the same line, type ; (a semicolon).

By Context

You use the F command (Find) to select a current line by context. For example,

```
F/Jabberwocky/
```

means to find the line containing 'Jabberwocky'. The search starts at the current line and moves forward through the source until the required line is found. If EDIT reaches the end of the source without finding a matching line, it displays the following message:

```
SOURCE EXHAUSTED
```

It is also possible to search backwards by using the BF command (Backwards Find). For example,

```
BF/gyre and gimble/
```

BF also starts with the current line, but EDIT moves backwards until it finds the desired line. If EDIT reaches the head of the output queue without finding a matching line, it displays the following message:

```
NO MORE PREVIOUS LINES
```

Notice that in the examples above, the desired text (Jabberwocky and gyre and gimble) is enclosed in matching single slashes (/). This desired text is called a **character string**. The characters you use to indicate the beginning and end of the character string are called **delimiter characters**. In the examples above, / was used as the delimiter. A number of special characters such as : . , and * are available for use as delimiters; naturally, the string itself must not contain the delimiter character. EDIT ignores the spaces between the command name and the first delimiter, but considers spaces within the string as significant, since it matches the context exactly. For example,

```
F /tum tum tree/
```

does not find 'tum-tum tree' or 'tum tum tree'.

If you use an F command with no argument, EDIT repeats the previous search. For example,

```
F/jubjub bird/; N; F
```

finds the second occurrence of a line containing 'jubjub bird'. The N command between the two F commands is necessary because an F command always starts by searching the current line. If you omitted N, the second F would find the same line as the first.

3.1.2.4 Qualifiers

The basic form of the F command described above finds a line that contains the given string anywhere in its length. To restrict the search to the beginning or the end of lines, you can place one of the letters B or E in front of the string. In this case, you must type one or more spaces after F. For example,

```
F B/slithy toves/
```

means Find the line Beginning with 'slithy toves', while

```
F E/bandersnatch/
```

means Find the line Ending with 'bandersnatch'. As well as putting further conditions on the context required, the use of B or E speeds up the search, as EDIT only needs to consider part of each line.

B and E as used above are examples of **qualifiers**, and the whole argument is called a **qualified string**. A number of other qualifiers are also available. For example,

```
F P/a-sitting on a gate/
```

means Find the next line containing Precisely the text 'a-sitting on a gate'. The required line must contain no other characters, either before or after the given string. That is to say, when you give this command, EDIT finds the next line containing:

```
a-sitting on a gate
```

However, EDIT does not find the line:

```
a-sitting on a gate.
```

To find an empty line (Precisely nothing), you can use an empty string with the P qualifier, for example,

```
F P//
```

You can give more than one qualifier in any order.

3.1.2.5 Making Changes to the Current Line

This section describes how to use the E, A, and B commands to alter the text on your current line.

Exchanging strings

The E command Exchanges one string of characters in the line for another, for example:

```
E/Wonderland/Looking Glass/
```

removes the string 'Wonderland' from the current line, and replaces it with 'Looking Glass'. Note that you use a single central delimiter to separate the two strings. To delete parts of the line (exchange text for nothing), you can use a null second string, as follows:

```
E/monstrous crow//
```

To add new material to the line, you can use the A or B commands. The A command inserts its second string After the first occurrence of the first string on the current line. Similarly, the B command inserts its second string Before the first occurrence of the first string on the current line. For example, if the current line contained

```
If seven maids with seven mops
```

then the following command sequence:

```
A/seven/ty/; B L/seven/sixty-/
```

would turn it into

```
If seventy maids with sixty-seven mops
```

If you had omitted the L qualifier from the B command above, the result would be

```
If sixty-seventy maids with seven mops
```

because the search for a string usually proceeds from left to right, and EDIT uses the first occurrence that it finds. You use the qualifier L to specify that the search should proceed Leftwards. The L qualifier forces the command that it qualifies to act on the Last occurrence of its first argument.

If the first string in an A, B or E command is empty, EDIT inserts the second string at the beginning or the end of the line. To further qualify the position of the second string, you use or omit the L or the E qualifiers.

If you give EDIT an A, B, or E command on a line that does not match the qualified string given as the first argument, the following message appears either on the screen or in a verification file that you specified when you entered EDIT.

```
NO MATCH
```

See the section "Calling EDIT" for details on the verification file.

3.1.2.6 Deleting Whole Lines

This section describes how to remove lines of text from your file. To delete a range of lines, you can specify their line numbers in a D command. To use the D command, type D and the line number. If you type a space and a second number after D, EDIT removes all the lines from the first line number to the last. For example,

```
D97 104
```

deletes lines 97 to 104 inclusive, leaving line 105 as the new current line. To delete the current line, type D without a qualifying number. For example,

```
F/plum cake/; D
```

deletes the line containing 'plum cake', and the line following it becomes the new current line. You can combine a qualified search with a delete command, as follows:

```
F B/The/; 4D
```

This command sequence deletes four lines, the first of which is the line beginning with 'The'.

You can also type a period (.) or an asterisk (*) instead of line numbers. To refer to the current line, type a period. To refer to the end-of-file, type an asterisk. For example,

```
D. *
```

deletes the rest of the source including the current line.

3.1.2.7 Inserting New Lines

This section describes how to insert text into your file with EDIT. To insert one or more lines of new material BEFORE a given line, you use the I command. You can give the I command alone or with a line number, a period (.), or an asterisk (*). EDIT inserts text before the current line if you give I on its own, or follow it with a period (.). If you type an asterisk (*) after I, your text is inserted at the end of the file (that is, before the end-of-file line). Any text that you type is inserted before the line you specified.

To indicate the end of your insertion, press RETURN, type Z, and press RETURN again. For example,

```
I 468
The little fishes of the sea,
They sent an answer back to me.
Z
```

inserts the two lines of text before line 468.

If you omit the line number from the command, EDIT inserts the new material before the current line. For example,

```
F/corkscrew/; I
He said, "I'll go and wake them, if..."
Z
```

This multiple command finds the line containing 'corkscrew' (which then becomes the current line) and inserts the specified new line.

After an I command containing a line number, the current line is the line of that number; otherwise, the current line is unchanged.

To insert material at the end of the file, type I*.

To save you typing, EDIT provides the R (Replace) command, the exact equivalent of typing DI (D for Delete followed by I for Insert). For example,

```
R19 26
In winter when the fields are white
Z
```

deletes lines 19 to 26 inclusive, then inserts the new material before line 27, which becomes the current line.

3.1.2.8 Command Repetition

You can also use individual repeat counts as shown in the examples for N and D above with many EDIT commands. In addition, you can repeat a collection of commands by forming them into a command group using parentheses as follows:

```
6(F P//; D)
```

deletes the next six blank lines in the source. Command groups may not extend over more than one line of command input.

3.1.3 Leaving EDIT

To end a EDIT session, you use the command W (for Windup). EDIT 'winds through' to the end of the source, copying it to the destination, and exits. Unless you specify a TO file, EDIT renames the temporary output file as the FROM filename.

EDIT can accept commands from a number of command sources. In the simplest case, EDIT accepts commands directly from the terminal (that is, from the keyboard); this is called the **primary command level**. EDIT can, however, accept commands from other sources, for example, **command files** or WITH files.

You can call command files from within EDIT, and further command files from within command files, with the C command, so that each nested command file becomes a separate command level. EDIT stops executing the commands in the command file when it comes to the end of the command file, or when it finds a Q. When EDIT receives a Q command in a command file, or it comes to the end of the file, it

immediately stops executing commands from that file, and reverts to the the previous command level. If EDIT finds a Q command in a nested command file, it returns to executing commands in the command file at the level above. If you stop editing at the primary command level, by typing Q, or if EDIT finds a Q in a WITH file, then EDIT winds up and exits in the same way as it does with W.

The command STOP terminates EDIT without any further processing. In particular, EDIT does not write out any pending lines of output still in memory so that the destination file is incomplete. If you only specify the FROM argument, EDIT does not overwrite the source file with the (incomplete) edited file. You should only use STOP if you do not need the output from the EDIT run.

EDIT writes a temporary backup to :T/ED-BACKUP when you exit with the W or Q commands. This backup file remains until you exit from EDIT with these commands again, whereupon EDIT overwrites the file with a new backup. If you use the STOP command, EDIT does not write to this file.

3.1.4 A Combined Example: Pulling It All Together

You can meet most simple editing requirements with the commands already described. This section presents an example that uses several commands. The text in italics following the editing commands in the example is a comment. You are not meant to type these comments; EDIT does not allow comments in command lines.

Take the following source text (with line numbers):

```
1  Tweedledee and Tweedledum
2  agreed to a battle,
3  For Tweedledum said Tweedledee
4  ad spoiled his nice new rattle.
5
6  As black as a tar barrel
7  Which frightened both the heroes so
8  They quite forgot their quorell
```

Execute these EDIT commands:

```
M1; E/dum/dee/; E/dee/dum/      the order of the
                                E commands matters!
N; E/a/A/; B /a /have /         now at line 2
F B/ad/; B//H/                  H at line start
F P//; N; I                     before line after blank one
Just then flew down a monstrous crow,
Z
M6; 2(A L//,/; N)               commas at end of lines
F/quore/; E/quorell/quarrel./
                                F is in fact redundant
W                               Windup
```

The following text (with new line numbers) is the result.

```
 1  Tweedledum and Tweedledee
 2  Agreed to have a battle,
 3  For Tweedledum said Tweedledee
 4  Had spoiled his nice new rattle.
 5
 6  Just then flew down a monstrous crow,
 7  As black as a tar barrel,
 8  Which frightened both the heroes so,
 9  They quite forgot their quarrel.
```

Note: If you experiment with editing this source file, you'll find that you don't have to use the commands in the example above. For instance, on the second line, you could use the following command:

```
E/a/have a/
```

to produce the same result. In other words, E Exchanges 'a' for 'have a', and B places 'have ' Before 'a' to produce 'have a'.

3.2 A Complete Specification of EDIT

After reading the first part of this chapter on the basic features of EDIT, you should be able to use the editor in a simple way. The rest of this chapter is a reference section that provides a full specification of all the features of EDIT. You may need to consult this section if you have any problems when editing or if you want to use EDIT in a more sophisticated way.

The features described in this section are as follows:

- Command syntax
- Control of Command, Input, and Output Files
- Processing EDIT
- Functional Groupings of EDIT Commands
- Line Windows
- String Operations on the Current Line
- Miscellaneous Current Line Commands
- Inspecting Parts of the Source: The Type Commands
- Control of Command, Input, and Output Files
- Loops
- Global Operations
- Displaying the Program State
- Terminating an EDIT Run
- Current Line Verification
- Miscellaneous Commands
- Abandoning Interactive Editing

3.2.1 Command Syntax

EDIT commands consist of a command name followed by zero or more arguments. One or more space characters may optionally appear between a command name and the first argument, between non-string arguments, and between commands. A space character is only necessary in these places to separate successive items otherwise treated as one (for example, two numbers).

EDIT understands that a command is finished in any of the following ways: when you press RETURN; when EDIT reaches the end of the command arguments; or when EDIT reads a semicolon (;), or closing parenthesis ( ), that you have typed.

You use parentheses to delimit command groups.

To separate commands that appear on the same line of input, you type a semicolon. This is only strictly necessary in cases of ambiguity where a command has a variable number of arguments. EDIT always tries to read the longest possible command.

Except where they appear as part of a character string, EDIT thinks of upper and lower case letters as the same.

3.2.1.1 Command Names

A command name is either a sequence of letters or a single special character (for example, #). An alphabetic command name ends with any non-letter; only the first four letters of the name are significant. One or more spaces may appear between command names and their arguments; EDIT requires at least one space when an argument starting with a letter follows an alphabetic name.

3.2.1.2 Arguments

The following sections describe the six different types of argument you can use with EDIT commands:

- strings
- qualified strings
- search expressions
- numbers
- switch values
- command groups

3.2.1.3 Strings

A string is a sequence of up to 80 characters enclosed in delimiters. You may use an empty (null) string. (A null string is exactly what it sounds like: a non-string, that is, delimiters enclosing nothing, for example, //.) The character that you decide to use to delimit a particular string may not appear in the string. The terminating delimiter may be omitted if it is immediately followed by the end of the command line.

The following characters are available for use as delimiters:

```
/  .  +  -  ,  ?  :  *
```

that is, common English punctuation characters (except ;) and the four arithmetic operators.

Here are some examples of strings:

```
/A/
*Menai Bridge*
??
+String with final delimiter omitted
```

3.2.1.4 Multiple Strings

Commands that take two string arguments use the same delimiter for both and do not double it between the arguments. An example is the A command:

```
A /King/The Red /
```

For all such commands the second string specifies replacement text. If you omit the second string, EDIT uses the null string. If you do this with the A and B command, then nothing happens because you have asked EDIT to insert nothing after or before the first string. However, if you omit the second string after an E command, EDIT deletes the first string.

3.2.1.5 Qualified Strings

Commands that search for contexts, either in the current line or scanning through the source, specify the context with qualified strings. A qualified string is a string preceded by zero or more qualifiers. The qualifiers are single letters. They may appear in any order. For example,

```
BU/Abc/
```

Spaces may not appear between the qualifiers. You may finish a list of qualifiers with any delimiter character. The available qualifiers are B (Beginning), E (End), L (Left or Last), P (Precisely), and U (Uppercase).

3.2.1.6 Search Expressions

Commands that search for a particular line in the source take a search expression as an argument. A search expression is a single qualified string. For example,

```
F B/Tweedle/
```

tells EDIT to look for a line beginning with the string 'Tweedle'.

3.2.1.7 Numbers

A number is a sequence of decimal digits. Line numbers are a special form of number and must always be greater than zero. Wherever a line number appears, the characters '.' and '*' may appear instead. A period represents the current line, and an asterisk represents the last line at the end of the source file. For example,

```
M*
```

instructs EDIT to move to the end of the source file.

3.2.1.8 Switch Values

Commands that alter EDIT switches take a single character as an argument. The character must be either a + or -. For example, in

```
V-
```

the minus sign (-) indicates that EDIT should turn off the verification. If you then type V+, EDIT turns the verification on again. In this case, you can consider + as 'on' and - as 'off'.

3.2.1.9 Command Groups

To make a number of individual EDIT commands into a command group, you can enclose them in parentheses. For example, the following line:

```
(f/Walrus/;e/Walrus/Large Marine Mammal/)
```

finds the next occurrence of 'Walrus' and changes it to 'Large Marine Mammal'. Command groups, however, may not span more than one line of input. For instance, if you type a command group that is longer than one line, EDIT only accepts the commands up to the end of the first line. Then, because EDIT does not find a closing parenthesis at the end of that line, it displays the following error message:

```
Unmatched parenthesis
```

Note that it is only necessary to use parentheses when you intend to repeat a command group more that once.

3.2.1.10 Command Repetition

EDIT accepts many commands preceded by an unsigned decimal number to indicate repetition, for example

```
24N
```

If you give a value of zero, then EDIT executes the command indefinitely (or until end-of-file is reached). For example, if you type

```
0(e /dum/dee/;n)
```

EDIT exchanges every occurrence of 'dum' for 'dee' to the end of the file.

You can specify repeat counts for command groups in the same way as for individual commands:

```
12(F/handsome/; E/handsome/hansom/; 3N)
```

3.2.2 Processing EDIT

This section describes what happens when you run EDIT. It gives details about where input comes from and where the output goes, what should appear on your screen, and what should eventually appear in your file after you have run EDIT.

3.2.2.1 Prompts

When EDIT is being run interactively, that is, with both the command file connected to the keyboard and the verification file connected to a window, it displays a prompt when it is ready to read a new line of commands. Although, if the last command of the previous line caused verification output, EDIT does not return a prompt.

If you turn the verification switch V on, EDIT verifies the current line in place of a prompt in the following circumstances:

- if it has not already verified the current line,
- if you have made any changes to the line since it was last verified, *or*
- if you have changed the position of the operational window.

Otherwise, when EDIT does not verify the current line, it displays a colon character (:) to indicate that it is ready for a new line of commands. This colon is the usual EDIT prompt.

EDIT never gives prompts when you are inserting lines.

3.2.2.2 The Current Line

As EDIT reads lines from the source file and writes them to the destination file, the line that EDIT has in its hand at any time is called the **current line**. Every command that you type refers to the current line. EDIT inserts new lines before the current line. When you start editing with EDIT, the current line is the first line of the source.

3.2.2.3 Line Numbers

EDIT identifies each line in the source by a unique line number. This is not part of the information stored in the file. EDIT computes these numbers by counting the lines as it reads them. EDIT does not assign line numbers to any new lines that you insert into the source.

EDIT distinguishes between original and non-original lines. Original lines are source lines that have not been split or inserted; non-original lines are split lines and inserted lines. Commands that take line numbers as arguments may only refer to original lines. EDIT moves forward, or backward up to a set limit, according to whether the line number you type is greater or less than the current line number. EDIT passes over or deletes (if appropriate) non-original lines in searches for a given original line.

When you type a period (.) instead of a line number, EDIT always uses the current line whether original or non-original. (For an example of its use, see Section 3.1.2.6, Deleting Whole Lines.)

You can renumber lines with the '=' command. This ensures that all lines following the current line are original. Type

```
=15
```

to number the current line as 15, the next line 16, the next 17, and so on to the end of the file. This is how you allocate line numbers to non-original lines. If you do not qualify the = command with a number, EDIT displays the message:

```
Number expected after =
```

3.2.2.4 Qualified Strings

To specify contexts for EDIT searches, you can use qualified strings. EDIT accepts the null string and always matches it at the initial search position, which is the beginning of the line except as specified below. In the absence of any qualifiers, EDIT may find the given string anywhere in a line. Qualifiers specify additional conditions for the context. EDIT recognizes five qualifiers B, E, L, P, and U as follows:

```
B
```

where the string must be at the Beginning of the line. This qualifier may not appear with E, L, or P.

```
E
```

where the string must be at the End of the line. This qualifier may not appear with B, L, or P. If E appears with the null string, it matches with the end of the line. (That is, look for nothing at the end of a line.)

```
L
```

where the search for the string is to take place Leftwards from the end of the line instead of rightwards from the beginning. If there is more than one occurrence of the string in a line, this qualifier makes sure that the Last one is found instead of the first. L may not appear with B, E, or P. If

L appears with the null string, it matches with the end of the line. (That is, look leftwards from the end of the line for an occurrence of nothing.)

```
P
```

where the line must match the string Precisely and must contain no other characters. P must not appear with B, E, or L. If P appears with a null string, it matches with an empty line.

```
U
```

where the string match is to take place whether or not upper or lower case is used. (That is, as though you translated both the string and the line into Uppercase letters before comparing them.) For example, when you specify U, the following string

```
/TWEEDledum/
```

should match a line containing

```
TweedleDUM
```

as well as any other combination in upper or lower case.

3.2.2.5 Output Processing

EDIT does not write lines read in a forward direction to the destination file immediately, but instead it adds them to an output queue in main memory. When EDIT has used up the memory available for such lines, it writes out the lines at the head of the queue as necessary. Until EDIT has actually written out a line to the destination file, you can move back and make it the current line again.

You can also send portions of the output to destination files other than TO. When you select an alternative destination file, EDIT writes out the queue of lines for the current output file.

3.2.2.6 End-of-File Handling

When EDIT reaches the end of a source file, a dummy end-of-file line becomes current. This end-of-file line has a line number one greater than the number of lines in the file. EDIT verifies the line by displaying the line number and an asterisk.

When the end-of-file line is current, commands to make changes to the current line, and commands to move forward, produce an error. Although, if you contain these kinds of commands within an infinitely repeating group, EDIT does not give an error on reaching the end-of-file line. The E (Exchange) command is an example of a command to make changes to the current line. The N (Next) command is an example of a command to move forward.

3.2.3 Functional Groupings of EDIT Commands

This section contains descriptions of all EDIT commands split up by function. A summary and an alphabetic list of commands appear later.

The following descriptions use slashes (/) to indicate delimiter characters (that is, characters that enclose strings). Command names appear in upper case; argument types appear in lower case as follows:

Notation	Description
a, b	line numbers (or . or *)
cg	command group
m, n	numbers
q	qualifier list (possibly empty)
se	search expression
s, t	strings of arbitrary characters
sw	switch value (+ or -)
/	string delimiter

Table 3.1 Notation for Command Descriptions

Note: Command descriptions that appear in the rest of this manual with the above notation show the SYNTAX of the command; they are not examples of what you actually type. Examples always appear as follows in

```
this typeface.
```

3.2.3.1 Selection of a Current Line

These commands have no function other than to select a new current line. EDIT adds lines that it has passed in a forward direction to the destination output queue (for further details on the output queue, see Section 3.1, Introducing EDIT). EDIT queues up lines that it has passed in a backward direction ready for subsequent reprocessing in a forward direction. M takes a line number, period, or asterisk. So, using the command notation described above, the correct syntax for M is as follows:

Ma

where Ma moves forward or backward to line 'a' in the source. Only original lines can be accessed by line number.

M+

makes the last line actually read from the file current line. M+ moves through all the lines currently held in memory until the last one is reached.

M-

makes the last line on the output queue current. This is like saying to EDIT: 'move back as far as you can.'

N

moves forward to the next line in the source. When the current line is the last line of the source, executing an N command does not create an error. EDIT increases the line number by adding one to it and creates a special

end-of-file line. However, if you try to use an N command when you are already at the end of the source file, EDIT returns an error.

P

moves back to the previous line. You can move more than one line back by either repeating P, or giving a number before it. The number that you give should be equal to the number of lines you want to move back.

The syntax for the F (Find) command is

F se

So, F finds the line you specify with the search expression 'se'. The search starts at the current line and moves forward through the source. The search starts at the current line in order to cover the case where the current line has been reached as a side effect of previous commands - such as line deletion. An F command with no argument searches using the last executed search expression.

The syntax for the BF (Backwards Find) command is

BF se

BF behaves like F except that it starts at the current line and moves backward until it finds a line that matches its search expression.

3.2.3.2 Line Insertion and Deletion

Commands may select a new current line as a side effect of their main function. Those followed by in-line insertion material must be the last command on a line. The insertion material is on successive lines terminated by Z on a line by itself. You can use the Z command to change the terminator. EDIT recognizes the terminator you give in either upper or lower case. For example, using the same notation,

```
Ia
<insertion material, as many
lines as necessary>
Z
```

inserts the insertion material before 'a'. Remember that 'a' can be a specific line number, a period (representing the current line), or an asterisk (representing the last line of the source file). If you omit a, EDIT inserts the material before the current line; otherwise, line a becomes the current line.

```
I /s/
```

inserts the contents of the file s (remember, 's' means any string) before the current line.

```
Ra b
<replacement material>
Z
Ra b /s/
```

The R command is equivalent to D followed by I. The second line number must be greater than or equal to the first. You may omit the second number if you want to replace just the one line (that is, if b = a). You may omit both numbers if you want to replace the current line. The line following line b becomes the new current line.

The syntax for the D (Delete) command is as follows:

```
Da b
```

So, D deletes all lines from a to b inclusive. You may omit the second line number if you want to delete just the one line (that is, if b = a). You may omit both numbers if you want to delete the current line. The line following line b becomes the new current line.

The syntax of the DF (Delete Find) command is

```
DF se
```

The command DF (Delete Find) tells EDIT to delete successive lines from the source until it finds a line matching the search expression. This line then becomes the new current line. A DF command with no argument searches (deleting as it goes) using the last search expression you typed.

3.2.4 Line Windows

EDIT usually acts on a complete current line. However, you can define parts of the line where EDIT can execute your subsequent commands. These parts of lines are called **line windows**. This section describes the commands you use to define a window.

3.2.4.1 The Operational Window

EDIT usually scans all the characters in a line when looking for a given string. However it is possible to specify a 'line window', so that the scan for a character starts at the beginning of the window, and not the start of the line. In all the descriptions of EDIT context commands, 'the beginning of the line' always means 'the beginning of the operational window'.

Whenever EDIT verifies a current line, it indicates the position of the operational window by displaying a '>' character directly beneath the line. For example in the following

```
26.
This is line 26 this is.
       >
```

the operational window contains the characters to the right of the pointer: 'line 26 this is.'. EDIT omits the indicator if it is at the start of the line.

The left edge of the window is also called the **character pointer** in this context, and the following commands are available for moving it:

```
>
```

moves the pointer one character to the right.

```
<
```

moves the pointer one character to the left.

```
PR
```

Pointer Reset sets the pointer to the start of the line.

The syntax for the PA (Point After) command is

```
PA q/s/
```

Point After sets the pointer so that the first character in the window is the first character following the string s. For example,

```
PA L//
```

moves the pointer to the end of the line.

The syntax for the PB (Point Before) command is

```
PB q/s/
```

Point Before is the same as PA, but includes the string itself in the window.

3.2.4.2 Single Character Operations on the Current Line

The following two commands move the character pointer one place to the right after forcing the first letter into either upper or lower case. If the first character is not a letter, or is already in the required case, these commands are equivalent to >.

The command

```
$
```

forces lower case (Dollar for Down).

The command

```
%
```

forces upper case (Percent for uP).

The '_' (underscore) command changes the first character in the window into a space character, then moves the character pointer one place to the right.

The command

```
#
```

deletes the first character in the window. The remainder of the window moves one character to the left, leaving the character pointer pointing at the next character in the line. The command is exactly equivalent to

```
E/s//
```

where s is the first character in the window. To repeat the effect, you specify a number before the '#' command. If the value is n, for example, then the repeated command is equivalent to the single command

```
E/s//
```

where s is the first n characters in the window or the whole of the contents of the window, whichever is the shorter. Consider the following example:

```
5#
```

deletes the next five characters in the window. If you type a number equal to or greater than the number of characters in the window, EDIT deletes the contents of the entire window. EDIT treats a sequence of '#' commands in the same way as a single, repeated '#' command. So, ##### is the same as typing a single #, pressing RETURN after each single #, five times.

You can use a combination of '>' '%' '$' '_' and '#' commands to edit a line character by character, the commands appearing under the characters they affect. The following text and commands illustrate this:

```
o Oysters,, Come ANDDWALK with    us
%>$$$$$$$#>>$$$$$$$_$$$$$$$$$###
```

The commands in the example above change the line to

```
O oysters, come and walk with us
```

leaving the character pointer immediately before the word 'us'.

3.2.5 String Operations on the Current Line

To specify which part of the current line to qualify, you can either alter the basic string or point to a variant, as described in the next two sections.

3.2.5.1 Basic String Operations

Three similar commands are available for altering parts of the current line. The A, B and E commands insert their second (string) argument After, Before, or in Exchange for their first argument respectively. You may qualify the first argument. If the current line were

```
The Carpenter beseech
```

then the commands

```
E U/carpenter/Walrus/   Exchange
B/bese/did /            Insert string before
A L//;/                 Insert string after
```

would change the line to

```
The Walrus did beseech;
```

3.2.5.2 The Null String

You can use the null, or empty string (//) after any string command. If you use the null string as the second string in an E command, EDIT removes the first string from the line. Provided EDIT finds the first string, an A or B command with a null second string does nothing; otherwise, EDIT returns an error. A null first string in any of the three commands matches at the initial search position. The initial search position is the current character position (initially the beginning of the line) unless either of the E or L qualifiers is present, in which case the initial position is the end of the line. For example,

```
A//carpenter/
```

puts the text carpenter After nothing, that is, at the beginning of the line. Whereas

```
A L//carpenter
```

puts carpenter at the end of the line After the Last nothing.

3.2.5.3 Pointing Variant

The AP (insert After and Point), BP (insert Before and Point), and EP (Exchange and Point) commands take two strings as arguments and act exactly like A, B, and E. However, AP, BP, and EP have an additional feature: when the operation is complete, the character pointer is left pointing to the first character following both text strings. So, using the same command syntax notation,

```
AP/s/t/
```

is equivalent to

```
A/s/t/; PA/st/
```

while

```
BP/s/t/
```

is equivalent to

```
B/s/t/; PA/ts/
```

and

```
2EP U/tweadle/Tweedle/
```

would change

```
tweadledum and TWEADLEdee
```

into

```
Tweedledum and Tweedledee
```

leaving the character pointer just before dee.

3.2.5.4 Deleting Parts of the Current Line

You use the commands DTA (Delete Till After) and DTB (Delete Till Before) to delete from the beginning of the line (or character pointer) to a specified string. To delete from a given context until the end of the line, you use the commands DFA (Delete From After) and DFB (Delete From Before). If the current line were

```
All the King's horses and all the King's men
```

then the command

```
DTB L/King's/
```

would change it to

```
King's men
```

while

```
DTA/horses /
```

would change it to

```
and all the King's men
```

3.2.6 Miscellaneous Current Line Commands

This section includes some further commands that explain how to repeat commands involving strings, how to split the current line, and how to join lines together.

3.2.6.1 Repeating the Last String Alteration

Whenever EDIT executes a string alteration command (for example, A, B, or E), it becomes the **current string alteration command**. To repeat the current string alteration command, you can type a single quote ('). The ' command has no arguments. It takes its arguments from the last A, B, or E command.

WARNING: Unexpected effects occur if you use sequences such as

```
E/castle/knight/; 4('; E/pawn/queen/)
```

The second and subsequent executions of the ' command refer to a different command than the first. The above example would exchange castle and knight twice and exchange pawn and queen seven times instead of exchanging castle and knight once and then four times exchanging castle and knight and pawn and queen.

3.2.6.2 Splitting and Joining Lines

EDIT is primarily a line editor. Most EDIT editing commands do not operate over line boundaries, but this section describes commands for splitting a line into more than one line and for joining together two or more successive lines.

To split a line before a specified context, you use the SB command. The syntax for the SB command is

```
SB q/s/
```

SB takes an optional qualifier represented here by q, and a string /s/. SB Splits the current line Before the context you specify with the qualifier and string. EDIT sends the first part of the line to the output and makes the remainder into a new, non-original current line.

To split a line after a specified context, you use the SA command. The syntax for SA is

SA q/s/

SA takes an optional qualifier and a string (q and /s/). SA Splits the current line After the context you specify with the qualifier and string.

To concatenate a line, you use the CL command. The syntax for CL is

CL/s/

CL takes an optional string that is represented here by /s/. CL or Concatenate Line forms a new current line by concatenating the current line, the string you specified and the next line from the source, in that order. If the string is a null string, you may type the command CL without specifying a string.

For an example of splitting and joining lines, look at the text

```
Humpty Dumpty sat on a wall; Humpty
Dumpty had a
great fall.
```

The old verse appears disjointed; the lines need to be balanced. If you make the first line the current line, the commands

```
SA /; /; 2CL/ /
```

change the line into

```
Humpty Dumpty sat on a wall;
```

leaving

```
Humpty Dumpty had a great fall.
```

as the new current line.

3.2.7 Inspecting Parts of the Source: the Type Commands

The following commands all tell EDIT to advance through the source, sending the lines it passes to the verification file as well as to the normal output (where relevant). Because these commands are most frequently used interactively (that is, with verification to the screen), they are known as the 'type' commands. They have this name because you can use them to 'type' out the lines you specify on the screen. This does not however mean that you cannot use them to send output to a file. After EDIT has executed one of these commands, the last line it 'typed' (that is, displayed) becomes the new current line.

The syntax for the T (Type) command is

```
Tn
```

Tn types n lines. If you omit n, typing continues until the end of the source. However, you can always interrupt the typing with CTRL-C.

Note: Throughout this manual when you see a hyphen between two keys, you press them at the same time. So CTRL-C means to hold down the CTRL key while you type C.

When you use the T command, the first line EDIT types is the current line, so that, for example,

```
F /It's my own invention/; T6
```

types six lines starting with the one containing 'It's my own invention'. (Note that to find the correct line, you must type the 'I' in 'It's' in upper case.)

The command

```
TP
```

types the lines in the output queue. Thus, TP (Type Previous) is equivalent to EDIT executing M- followed by typing until it reaches the last line it actually read from the source.

The command

TN

types until EDIT has changed all the lines in the output queue. (For more information on the output queue, see Section 3.1, Introducing EDIT) So, a TN (Type Next) command types N lines, where N was the number specified as the P option. (To find out more about the P option, refer to Section 3.1.1, Calling EDIT). The advantage of the TN command is that everything visible during the typing operation is available in memory to P and BF commands.

The syntax for the TL (Type with Line numbers) command is as follows:

TLn

TLn types n lines as for T, but with line numbers added. Inserted and split lines do not have line numbers, EDIT displays a '+ + + +' instead. For example,

```
20   O oysters, come and walk with us
++++ and then we'll have some tea
```

The original line starting with 'O oysters' has a line number. The non-original line, inserted after line 20, starts with + + + +. (Remember that you can use the = command to renumber non-original lines.)

3.2.8 Control of Command, Input and Output Files

EDIT uses four types of files:

- command
- input
- output
- verification

Once you have entered EDIT, you cannot change the verification file with a command. (To find out more about the verification file, see Section

3.1.1, Calling EDIT.) The following sections describe commands that can change the command, input, and output files that you set up when you enter EDIT.

3.2.8.1 Command Files

When you enter EDIT, it reads commands from the terminal or the file that you specify as WITH. To read commands from another file, you can use the C command. The syntax for the command is

C .s.

where the string 's' represents a filename. As Tripos uses the slash symbol (/) to separate filenames, you should use periods (.), or some other symbol, to delimit the filename. A symbol found in a string should not be used as a delimiter. When EDIT has finished all the commands in the file (or you give a Q command), it closes the file and control reverts to the command following the C command. For example, the command

```
C .:T/XYZ.
```

reads and executes commands from the file :T/XYZ

3.2.8.2 Input Files

To insert the entire contents of a file at a specific point in the source, you use the I and R commands. These commands are described in Section 3.1.2.7 earlier in this chapter.

Section 3.1.1 described how to call EDIT. In that section, the source file was referred to as the FROM file. However, you can also associate the FROM file with other files, using the command FROM. The FROM command has the following form:

FROM .s.

where the string 's' is a filename. A FROM command with no argument re-selects the original source file.

When EDIT executes a FROM command, the current line remains current; however, the next line comes from the new source.

EDIT does not close a source file when the file ceases to be current; you can read further lines from the source file by re-selecting it later.

To close an output file that you opened in EDIT, and that subsequently you want to open for input (or the other way round), you must use the CF (Close File) command. The CF command has the following form:

```
CF .s.
```

where the string 's' represents a filename. When you end an EDIT session, EDIT closes automatically all the files you opened in EDIT.

Note: Any time you open a file, EDIT starts at the beginning of that file. If you close a file with CF, EDIT starts on the first line of that file if you re-open it, and not at the line it was on when you closed the file.

An example of the use of the FROM command to merge lines from two files follows:

Command	**Action**
`M10`	*Pass lines 1-9 from the FROM (source) file*
`FROM .XYZ.`	*Select new input, line 10 remains current*
`M6`	*Pass line 10 from FROM, lines 1-5 from XYZ*
`FROM`	*Reselect FROM*
`M14`	*Pass line 6 from XYZ, lines 11-13 from FROM*
`FROM .XYZ.`	*Reselect XYZ*
`M*`	*Pass line 14 from FROM, the rest of XYZ*
`FROM`	*Reselect FROM*
`CF .XYZ.`	*Close XYZ*
`M*`	*Pass the rest of FROM (lines 15 till end-of-file)*

3.2.8.3 Output Files

EDIT usually sends output to the file with filename TO. However, EDIT does not send the output immediately. It keeps a certain number of lines in a queue in main memory as long as possible. These lines are previous current lines or lines that EDIT has passed before reaching the present current line. The number of lines that EDIT can keep depends on the options you specified when you called EDIT. Because EDIT keeps these lines, it has the capability for moving backwards in the source.

To associate the output queue with a file other than that with the filename TO, you can also use the TO command. The TO command has the form

TO .s.

where s is a filename.

When EDIT executes a TO command, it writes out the existing queue of output lines if the output file is switched.

EDIT does not close an output file when it is no longer current. By re-selecting the file, you can add further lines to it. The following example shows how you can split up the source between the main destination TO and an alternate destination XYZ.

Command	**Action**
M11	*Pass lines 1-10 to TO*
TO.XYZ.	*Switch output file*
M21	*Pass lines 11-20 to XYZ*
TO	
M31	*Pass lines 21-30 to TO*
TO.XYZ.	
M41	*Pass lines 31-40 to XYZ*
TO	

If you want to re-use a file, you must explicitly close it. The command

```
CF .filename.
```

closes the file with the filename you specify as the argument.

These input/output commands are useful when you want to move part of the source file to a later place in the output. For example,

Command	Action
`TO .:T/1.`	*Output to temporary file*
`1000N`	*Advance through source*
`TO`	*Revert to TO*
`CF .:T/1.`	*Close output file :T.1*
`I2000.:T/1.`	*Re-use as input file*

If you use the CF command on files you have finished with, the amount of memory you need is minimized.

3.2.9 Loops

You can type an unsigned decimal number before many commands to indicate repetition, for example,

```
24N
```

You can also specify repeat counts for command groups in the same way as for individual commands, for example,

```
12(F/handsome/; E/handsome/hansom/; 3N)
```

If you give a repeat count of zero (0), the command or command group is repeated indefinitely or until EDIT reaches the end of the source.

3.2.10 Global Operations

Global operations are operations that take place automatically as EDIT scans the source in a forward direction. You can start and stop global operations with special commands, described in the following sections.

WARNING: Be careful when you move backwards through the source not to leave any active or 'enabled' globals. These enabled globals could undo a lot of your work!

3.2.10.1 Setting Global Changes

Three commands, GA, GB, and GE are provided for simple string changes on each line. Their syntax is as follows:

```
GA q/s/t/
GB q/s/t/
GE q/s/t/
```

These commands apply an A, B or E command, as appropriate, to any occurrence of string 's' in a new current line. They also apply to the line that is current at the time the command is executed.

G commands do not re-scan their replacement text; for example, the following command

```
GE/Tiger Lily/Tiger Lily/
```

would not loop forever, but would have no visible effect on any line. However, as a result of the 'change', EDIT would verify certain lines.

EDIT applies the global changes to each new current line in the order in which you gave the commands.

3.2.10.2 Cancelling Global Changes

The REWIND command cancels all global operations automatically. You can use the CG (Cancel Global) command to cancel individual commands at any time.

When a global operation is set up by one of the commands GA, GB, or GE, the operation is allocated an identification number which is output to the verification file (for example, G1). The argument for CG is the number of the global operation to be cancelled. If CG is executed with no argument, EDIT cancels all globals.

3.2.10.3 Suspending Global Changes

You can suspend individual global operations, and later resume using them with DG (Disable Global) and EG (Enable Global) commands. These take the global identification number as their argument. If you omit the argument, all globals are turned off or on (disabled or enabled), as appropriate.

3.2.11 Displaying the Program State

Two commands beginning with SH (for SHow) output information about the state of EDIT to the verification file.

The command SHD (SHow Data) takes the form

```
SHD
```

and displays saved information values, such as the last search expression.

The command SHG (SHow Globals) takes the form

```
SHG
```

and displays the current global commands, together with their identification numbers. It also gives the number of times each global search expression matches.

3.2.12 Terminating an EDIT Run

To 'wind through' the rest of the source, you use the W command (Windup). Note that W is illegal if output is not currently directed to TO. EDIT exits when it has reached the end of the source, closed all the files, and relinquished the memory. Reaching the end of the highest level command file has the same effect as W. If you call EDIT specifying only the FROM filename, EDIT renames the temporary output file it created with the same name as the original (that is, the FROM filename), while it renames the original information as the file :T/EDIT-BACKUP. This backup file is, of course, only available until the next time EDIT is run.

The STOP command stops EDIT immediately. No further input or output is attempted. In particular, the STOP command stops EDIT from overwriting the original source file. Typing STOP ensures that no change is made to the input information.

The Q command stops EDIT from executing the current command file (EDIT initially accepts commands from the keyboard, but you can specify a command file with the WITH keyword or with the C command) and makes it revert to the previous one. A Q at the outermost level does the same as a W.

3.2.13 Current Line Verification

The following circumstances can cause automatic verification to occur:

- When you type a new line of commands for a current line that EDIT has not verified since it made the line current, or changed since the last verification.

- When EDIT has moved past a line that it has changed, but not yet verified.

- When EDIT displays an error message.

In the first two cases, the verification only occurs if the V switch is on. The command

V sw

changes the setting of the V switch. It is set ON (V+) if the initial state of EDIT is interactive (commands and verifications both connected to a terminal), and to OFF (V-) otherwise.

To explicitly request verification of the current line, you use the following command

?

This command verifies the current line. It is performed automatically if the V switch is on and the information in the line has been changed. The verification consists of the line number (or ++++ if the line is not original), with the text on the next line.

An alternate form of verification, useful for lines containing non-printing characters, is provided by the command

!

The ! command verifies the current line with character indicators. EDIT produces two lines of verification. The first is the current line in which EDIT replaces all the non-graphic characters with the first character of their hexadecimal value. In the second line, EDIT displays a minus sign under all the positions corresponding to uppercase letters and the second hexadecimal digit in the positions corresponding to non-graphic characters. All other positions contain space characters.

The following example uses the ? and ! commands. To verify the current line, you use the ? command. If, for instance, the following appears when you use the ? command:

```
?
1.
The Walrus and the ??
```

then you might try to use the E command to exchange '??' for 'Carpenter'. However, EDIT may not recognize the text it displayed with '??' as two question marks if the '??' characters correspond to two non-graphic characters. To find out what really is there, you use the ! command as follows:

```
!
1.
The Walrus and the ll
-   -              44
```

To correct the line, you can use the character pointer and # command to delete the spurious characters before inserting the correct text. (For further details on using the character pointer and # command, see Section 3.2.4, Line Windows.)

3.2.14 Miscellaneous Commands

This section describes all those commands that do not fit neatly into any of the previous categories. It describes how to change a termination character, turn trailing spaces off, renumber lines, and rewind the source file.

To change the terminator for text insertion, you use the Z command. The Z command has the following form

Z /s/

where /s/ represents a string. The string may be of any length up to 16 characters. The string is matched in either case. In effect, the search for the terminator is done using the qualifiers PU. The initial terminator string is Z.

To turn trailing spaces on or off, you use the TR (TRailing spaces) command. The TR command takes the following form

TR sw

where sw represents a switch (+ for ON; - for OFF). EDIT usually suppresses all trailing spaces. TR+ allows trailing spaces to remain on both input and output lines.

To renumber the source lines, you use the = command. The = command takes the form

=n

where n represents a number. The command =n sets the current line number to n. If you then move to the lines below the current line, EDIT renumbers all the following original and non-original lines. Although, if you move back to previous lines after using the = command, EDIT marks all the previous lines in the output queue as non-original. When you rewind the source file, EDIT renumbers all the lines in the file - original, non-original, and those previously re-numbered with the = command.

To rewind the source file, you use the REWIND command. For example,

REWIND

This command rewinds the input file so that line 1 is the current line again. First EDIT scans the rest of the source (for globals, and so forth). Then it writes the lines to the destination, which is then closed and re-opened as a new source. It closes the original source using a temporary file as a destination. Any globals that you specify are cancelled. EDIT does not necessarily require you to type the complete word (that is, REWIND). To move to the beginning, you can type any of the following: REWI, REWIN, or REWIND.

3.2.15 Abandoning Interactive Editing

To abandon most commands that read text, you press CTRL-C. In particular, if you realize that a search expression has been mistyped, then CTRL-C stops the search. Similarly the T command types to the end of the source, but CTRL-C abandons this action.

After you press CTRL-C, EDIT responds with the message

```
*** BREAK
```

and returns to reading commands. The current line does, of course, depend on exactly when you pressed CTRL-C.

Quick Reference Card

This list uses the following abbreviations:

Notation	Description
qs	Qualified string
t	String
n	Line number, or . or * (current and last line)
sw	+ or - (on or off)

Character Pointer Commands (Line Window Commands)

Command	Action
<	Move character pointer left
>	Move character pointer right
#	Delete character at pointer
$	Lower case character at pointer
%	Upper case character at pointer
_	Turn character at pointer to space
PA qs	Move character pointer to after qs
PB qs	Move character pointer to before qs
PR	Reset character pointer to start of line

Positioning Commands

Command	Action
M n	Move to line n
M +	Move to highest line in memory
M -	Move to lowest line in memory
N	Next line
P	Previous line
REWIND	Rewind input file

Search Commands

Command	Action
F qs	Find string qs
BF qs	Same as F, but move backwards through file
DF qs	Same as F, but delete lines as they are passed

Text Verification

Command	Action
?	Verify current line
!	Verify with character indicators
T	Type to end of file
T n	Type n lines
TL n	Type n lines with line numbers
TN	Type until buffer changed
TP	M-, then type to last line in buffer
V sw	Set verification on or off

Operations on the Current Line

Command	Action
A qs t	Place string t after qs
AP qs t	Same as A, but move character pointer
B qs t	Place string t before qs
BP qs t	Same as B, but move character pointer
CL t	Concatenate current line, string t and next line
D	Delete current line
DFA qs	Delete from after qs to end of line
DFB qs	Delete from before qs to end of line
DTA qs	Delete from start of line to after qs
DTB qs	Delete from start of line to before qs
E qs t	Exchange string qs with string t
EP qs t	Same as E, but move character pointer
I	Insert material from terminal before line
I t	Insert from file t
R	Replace material from terminal
R t	Replace material from file t
SA qs	Split line after qs
SB qs	Split line before qs

Globals

Command	Action
GA qs t	Globally place t after qs
GB qs t	Globally place s before qs
GE qs t	Globally exchange qs for t
CG n	Cancel global n (all if n omitted)
DG n	Disable global n (all if n omitted)
EG n	Enable global n (all if n omitted)
SHG	Display info on globals used

Input/Output Manipulation

Command	Action
FROM	Take source from original
FROM t	Take source from file t
TO	Revert to original destination
TO t	Place output lines in file t
CF t	Close file t

Other Commands

Command	Action
'	Repeat previous A, B or E command
= n	Set line number to n
C t	Take commands from file t
H n	Set halt at line n. If n = * then halt and unset h
Q	Exit from command level; windup if at level 1
SHD	Show data
STOP	Stop
TR sw	Set/unset trailing space removal
W	Windup
Z t	Set input terminator to string t

Appendix A: Error Codes and Messages

The error messages that appear on the screen when you use the FAULT or WHY command fall into two general categories:

1. user errors

2. programmer errors.

This appendix gives the probable cause and a suggestion for recovery for each of these error codes. The codes appear in numerical order within their category.

User Errors

103: insufficient free store

Probable cause:

You don't have enough physical memory on the computer to carry this operation out.

Recovery suggestion:

First, try to stop some of the applications that are running that you don't need. For example, close any unnecessary I/O streams. Otherwise, buy more memory. Stop some of the tasks that are less important to you and re-issue the command. It may be that you have enough memory, but it has become 'fragmented'; rebooting may help.

104: task table full

Probable cause:

Limited to 20 CLI tasks, or equivalent.

120: argument line invalid or too long

Probable cause:

Your argument for this command is incorrect or contains too many options.

Recovery suggestion:

Consult the command specifications in Chapter 1 of this manual for the correct argument template.

121: file is not an object module

Probable cause:

Either you misspelled the command name, or this file may not be in loadable file form.

Recovery suggestion:

Either retype the file name, or make sure that the file is a binary program file. Remember that in order to execute a command sequence the command C must be used before the file name.

202: object in use

Probable cause:

The file or directory specified is already being used by another application in a manner incompatible with the way you want to use it.

Recovery suggestion:

If another application is writing to a file, then nobody else can read from it. If another application is reading from a file, then nobody else can write to it. If an application is using a directory or reading from a file, then nobody else may delete or rename the file or directory. You must stop the other application using the file or directory and then try again.

203: object already exists

Probable cause:

The object name that you specified is that of an object that already exists.

Recovery suggestion:

You must first delete the directory or file if you really want to re-use that name.

204: directory not found

Probable cause:

The directory you specified is not in the current directory. This can happen if you misspell the directory name or omit the correct path.

Recovery suggestion:

Use LIST to check on the spelling of the directory name and then retype the command line with the correct spelling. Otherwise, include the correct path before the directory name.

205: object not found

Probable cause:

Tripos cannot find the device or file you specified. You have probably made a typographical or spelling error.

Recovery suggestion:

Check device names and file names for correct spellings. You also get this error if you attempt to create a file in a directory that does not exist.

210: invalid stream component name

Probable cause:

You have included an invalid character in the filename you have specified, or the filename is too long. Each file or directory must be less than 30 characters long. A filename cannot contain control characters.

212: object not of required type

Probable cause:

Maybe you've tried to do an operation that requires a filename and you gave it a directory name or *vice versa*. For example, you might have given the command TYPE dir, where 'dir' is a directory. Tripos doesn't allow you to display a directory, only files.

Recovery suggestion:

Check on the command usage in Chapter 1 of the *Tripos User's Reference*.

213: disk not validated

Probable cause:

Either you just inserted a disk and the disk validation process is in progress, or it may be a bad disk.

Recovery suggestion:

Wait for the disk validation process to finish - it normally only takes less than a minute. If Tripos cannot validate the disk because it is bad, then the disk remains unvalidated. In this case, you can only read from the disk and you must copy your information onto another disk.

214: disk write-protected

Probable cause:

This disk is write-protected. The computer cannot write over information that is already on the disk. You can only read information from this disk. You cannot store any information of your own here.

Recovery suggestion:

Save your information on a disk that is not write-protected, or change the write-protect tab on the disk.

215: rename across devices attempted

Probable cause:

RENAME only changes a filename on the same device, although you can use it to rename a file from one directory into another on the same device.

Recovery suggestion:

Copy the file to the object device and delete it from the source device.

216: directory not empty

Probable cause:

You cannot delete a directory unless it is empty.

Recovery suggestion:

Delete the contents of the directory. Study the command specification for DELETE in Chapter 1 of this manual.

218: device not mounted

Probable cause:

The word 'mounted' here means "inserted into the drive"; either you've made a typographical error, or the disk with the desired name isn't mounted.

Recovery suggestion:

Check the spelling of the devices, or insert the correct disk.

220: comment too big

Probable cause:

Your filenote has exceeded the maximum number of characters allowed (80).

Recovery suggestion:

Retype the filenote adhering to these limits.

221: disk full

Probable cause:

You do not have sufficient room on the disk to do this operation.

Recovery suggestion:

Use another disk or delete some unnecessary files or directories.

222: file is protected from deletion

Probable cause:

The file or directory has been protected from deletion.

Recovery suggestion:

You either did not mean to delete that file, or you really did mean it. If you really did mean it, you must use the PROTECT command to alter the protection status. Refer to the PROTECT command in the *Tripos User's Reference*. Also use the LIST command to check on what the protections of this particular file or disk.

225: not a DOS disk

Probable cause:

The disk in the drive is not a formatted DOS disk.

Recovery suggestion:

Place a suitably formatted DOS disk in the drive instead, or else format the disk using the FORMAT command if you don't want any of the information on it.

226: no disk in drive

Probable cause:

You have attempted to read or write to a disk drive where there is no disk.

Recovery suggestion:

Place a suitably formatted DOS disk in the drive.

Programmer Errors

209: packet request type unknown

Probable cause:

You have asked a device handler to attempt an operation it cannot do (for example, the console handler cannot rename anything).

Recovery suggestion:

Check the request code passed to device handlers.

211: invalid object lock

Probable cause:

You have used something that is not a valid lock.

Recovery suggestion:

Check your code so that you only pass valid locks to DOS calls that expect locks.

219: seek error

Probable cause:

You have attempted to call Seek with invalid arguments.

Recovery suggestion:

Make sure that you only Seek within the file. You cannot Seek to outside the bounds of the file.

232: no more entries in directory

Probable cause:

There are no more entries in the directory that you are examining.

Recovery suggestion:

This error code indicates that the DOS call ExNext has no more entries in the directory you are examining to hand back to you. Stop calling ExNext.

Tripos Programmer's Reference Manual

This manual refers to Issue 5, May 1986

Printed in the U.K

DISCLAIMER

THIS PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT THE DEVELOPER OR METACOMCO PLC OR ITS AFFILIATED DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, METACOMCO PLC OR ITS AFFILIATED COMPANIES DO NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK. IN NO EVENT WILL METACOMCO PLC OR ITS AFFILIATED COMPANIES BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Tripos Programmer's Reference Manual

Issue 5 (May 1986)

Table of Contents

Chapter 1: Introduction to Programming

This section of the manual describes some of the principles involved in writing programs to run under the Tripos operating system.

1.1 Programming under Tripos

There are two ways in which a program can be run under Tripos: as a command running under a CLI (Command Line Interpreter), or as a task (process). In either case the environment is very similar, although you should remember that a program running under the CLI is part of the CLI task. If you run your program as a separate task, then it will have no associated CLI or CLI variables.

Whatever method you choose, you must first create an executable program, which will almost certainly call the Tripos resident functions described in the next two chapters.

Note: The terms 'task' and 'process' are synonymous in Tripos.

1.1.1 Creating an Executable Program

A program may be written in any of the supported language translators running under Tripos (for example, C or Pascal). These translators all produce Tripos Binary Format files, which are provided as input to the linker ALINK. The linker takes these files as input, possibly scans a library, and produces Tripos Load Format files as output. This means that even if you have written a program in assembler that needs no runtime library support or combination with any other program, you must still pass the output of the assembler through ALINK to convert the file format. Besides scanning the language support library and the Tripos system library, ALINK requires you to specify a startup file followed by one or more user-program files, and this is true for linking programs in most programming languages under Tripos.

1.1.2 Calling Tripos Functions

Two major parts of Tripos are the Kernel and the DOS. There is no difference in calling either the Kernel or the DOS if you program in C or BCPL, as the Tripos system library provides a set of interface routines. So long as you use the standard C initialization code then all the calls are available to you.

If you wish to program in assembler, you must be aware of the different call sequences used. To call the Kernel, place the arguments required in registers D1 to D4 and a function code in D0, and code a TRAP #0. Tripos then performs the function, returns the result in register D0 and a secondary result in D1, while preserving the other registers.

To call the DOS, you need to know the library base pointer for the DOS. To obtain this, call the Kernel routine FindDOS which returns the pointer for you. Place this pointer into any convenient address register and then call the offset from the base pointer corresponding to the DOS function that you need. A set of names is provided in the standard header file 'tripos.i'. These names specify the offsets for the DOS calls as well as the Kernel function codes.

A full example of assembly language programming is at the end of this chapter.

1.1.3 Running Programs under the CLI

When you load a program under a CLI you type the name of the program and a set of arguments. You may also specify input or output redirection with the '<' and '>' symbols. The CLI sets up two initial file handles for the program that represent the standard input and output. Unless you redirect them, the standard input is from the keyboard and the standard output is to the screen.

When the CLI starts a program, it allocates a stack for that program. This stack is initially 4000 bytes but you may change the stack size with the STACK command. Tripos obtains this stack from the general free memory pool (sometimes referred to as the 'heap') just before you run a program; it is not, however, the same stack as the CLI uses. Tripos

pushes a suitable return address onto the stack that points to a routine to tell the CLI to regain control and unload your program. Below this on the stack at 4(SP) is the size of the stack in bytes, which may be useful if you wish to perform stack checking.

Your program starts with register A0 pointing to the arguments specified after the command name. Tripos stores the argument line in memory within the CLI stack and this pointer remains valid throughout your program. Register D0 indicates the number of characters in the argument line. You can use these initial values to decode the argument line to find out what the user requires. If you are programming in C, then the standard C startup file does this work for you and decodes the arguments into argv and the number of arguments into argc.

While your program is running, it may corrupt any register. Tripos function calls only corrupt the result registers D0 and D1.

To gain access to the two initial file handles that are set up by the CLI, you call the DOS functions Input and Output. Remember that you must find the DOS base pointer first; this can be done by calling the Kernel function FindDOS. You should not close these file handles within your program; the CLI opened them on your behalf and it will close them again, if required. Again, if you are programming in C, the initial file handles are obtained for you in the startup code and are placed into the variables stdin and stdout.

1.1.4 Running Programs as a New Task

There are a number of small differences between writing a program to run under a CLI and writing one to run as a separate task (process). There are only two ways to create a distinct new task. The first way is to write a device handler task that is dynamically created when referenced. To do this you need to use the MOUNT command to set up a reference to your new task. Writing a device handler task is a specialist activity, and it is described in more detail in the *Tripos Technical Reference Manual*.

The second and more common way to create a new task is to generate one from another program loaded as a CLI command. In this case the CLI program uses LoadSeg to load the code for the new task, and then creates a new task (process) by calling CreateProc. To set the new task running, the CLI command then has to send it an initial packet. Although the CLI command might then terminate, the new task keeps running until Exit is called.

A program running as a task finds a suitable return address pushed onto the stack, as well as the stack size on the stack below that. This is the same as when you run a program under a CLI. The task is initialized to a state such that it had just called TaskWait, so that it won't proceed until it receives a packet. This initial packet is normally sent by the task that created the new one, and may contain parameters for the new task. Usually the startup packet is sent back as soon as possible with a return code in the Res1 field of the packet to indicate whether the initialization worked or not. The startup packet may also pass other information, such as pointers to shared workspace, and so on.

It is important to note that a program running as a new task has no default input and output streams. You must ensure that your program opens any I/O channels that it needs, and that it closes them when it terminates.

1.1.5 Program Termination

To exit from a program running under the CLI or as a task, it is sufficient to give a simple RTS using the initial stack pointer (SP). In this case, you should also provide a return code in D0. This is zero if your program succeeded, or a positive number otherwise. If you return a non-zero number then the CLI notices an error. Depending on the current fail value (set by the command FAILAT), a non-interactive CLI, such as one running a command sequence set up by the Tripos C command, terminates. A program written in the C programming language can simply return from main(), which returns to the startup code, clears D0, and performs an RTS.

Alternatively, a program may call the Tripos function Exit which takes the return code as argument. This instructs your program to exit no matter what value the stack pointer has.

It is important to stress at this stage that Tripos does not control any resources; this is left entirely to the programmer. Any files that a user program opens must be closed before the program terminates. Likewise, any locks it obtains must be freed, any code it loads must be unloaded, and any memory allocated returned to the free pool. Of course, there may well be cases where you do not wish to return all the resources you obtained. For example, you may have written a program that loads a code segment into memory for later use. This is perfectly acceptable, but you need to have a mechanism for returning any memory, file locks, and so on. For a normal program a common technique is to provide a standard tidyup routine that puts back any resources and then calls Exit(). You then always call tidyup rather than Exit directly.

1.1.6 Example

This simple example prints a prompt to the terminal, waits for the user to type a response, then prints another message, and echos the response.

To start with, include a standard header file that contains useful definitions, including the offsets from the base pointer for each DOS function and the function codes for the Kernel calls:

```
INCLUDE "tripos.i"
```

The values of the offsets have the same name as that described in Chapter 3, "Calling the DOS," but each one is preceded by __LVO (Library Vector Offset) to distinguish it from the C function entry point.

Set up a macro to call the DOS to make the program more readable:

```
CALL    MACRO
        JSR         _LVO\1(A2)
        ENDM
```

You are now ready to start the program. First, find the DOS base pointer by calling the Kernel via a TRAP instruction:

```
MOVEQ     #K_FindDOS,D0 Kernel function code
TRAP      #0            No arguments
MOVEA.L   D0,A2         DOS base pointer in A0
```

You'll need to locate the channels that represent the standard input and output. If the program has been invoked under the Command Line Interface (CLI), then the calls Input and Output return the file handles associated with these channels. If you run the program as a task, you'll need to open a specific I/O channel. This complexity is not covered here. Next, save the file handles in D7 and D6:

```
CALL      Input
MOVE.L    D0,D7         D7 holds stdin
CALL      Output
MOVE.L    D0,D6         D6 holds stdout
```

Load the string for the prompt into A0 and branch to a small subroutine that prints the text held in A0. This example uses the convention that the length of the string is held in the initial byte pointed at by A0.

```
LEA.L     PROMPT,A0     String ptr
BSR.S     WRSTR         Print message
```

Next, read in the string typed by the user. Set A3 to point to the buffer for the characters, and then place this in D2 ready for the read. Add one to the buffer pointer to skip the length count that you are going to place in the initial byte. The file handle for the standard input comes from D6 into D1 for the call, and D3 holds the buffer size. Then call Read to read the characters from the keyboard. The operating system call returns when the buffer is full or when the user presses RETURN. Tripos also handles character reflection and rubout. When the call returns, the number of characters read is in D0, and this is placed in the byte pointed at by A3. This ensures that the string just entered has the same format as your other strings with the length of the string in the first byte.

```
        LEA.L   BUFFER,A3       Get name buffer
        MOVE.L  D7,D1           Use stdin handle
        MOVE.L  A3,D2           Read buffer pointer
        ADDQ.L  #1,D2           Skip length byte
        MOVEQ   #30,D3          Maximum name size
        CALL    Read            Fetch name
        MOVE.B  D0,(A3)         Insert length
```

Now print the next two strings. The first is the standard message; the second is the string just read from the keyboard.

```
        LEA.L   MESS,A0         String ptr
        BSR.S   WRSTR           Print message
        LEA.L   BUFFER,A0       Ptr to name
        BSR.S   WRSTR           Print message
```

To terminate the program, call the Tripos function Exit. This function exits the program with the return code in D1. Use a non-zero return code if you cannot open the Tripos library.

```
        MOVEQ   #0,D1           Zero return code
EX      CALL    Exit

ERR     MOVEQ   #20,D1          Non zero return code
        BRA.S   EX
```

This subroutine is used to print a string on the console. A string in this case is represented by a byte containing the length followed by the characters involved. A0 is the pointer to the string and D7 holds the file handle for the standard output, which is moved to D1 for the call. The length byte is extracted and placed into D3, while the updated value of A0 is used as the buffer pointer in D2. You can then call Write to do the work for you and finally return to the caller.

```
WRSTR   MOVE.L    D6,D1          Stdout file handle
        MOVEQ     #0,D3          Clear length register
        MOVE.B    (A0)+,D3       Get byte length of string
        MOVE.L    A0,D2          String is write buffer
        CALL      Write
        RTS
```

The data statements define the space for the character buffer and the prompt strings, which are preceded by a length byte.

```
        DATA
BUFFER  DS.B      1
        DS.B      30
MESS    DC.B      6,'Hello '
PROMPT  DC.B      18,'Enter your name : '
        END
```

1.2 Tasks

Tripos tasks fall into four main types

- device handler
- CLI
- debug
- user tasks

The first two are always present, whereas the third and fourth are usually, but not always, included. All tasks are, in fact, identical in construction, although they are different in what they do. All four types of task are described below.

1.2.1 Device Handler Tasks

Very few application programs communicate directly with a device driver, instead they communicate through a **device handler** and the device handler then talks to the device driver.

Some device handlers are really very simple as all they do is provide the standard set of Read and Write, Open and Close calls for the device they are associated with. For example, the serial line handler task takes Read requests and passes them on to the serial device and, in the same way, it accepts and handles Write requests. More importantly, a device handler task stops two programs trying to use the serial line at the same time.

At the other end of the spectrum is the file system task which, again, is a device handler task. This is really quite complicated because it provides a set of Open, Read, and Write functions for files, supports the hierarchical file system of Tripos, and maps all of this to calls to Read and Write sectors off the disk via the device driver.

There are many task handlers: the serial line task handler, the parallel port task handler, file system task handler, and the console task handler. The console task supports the standard console, which, if it is connected to a serial line, supports character reflection, rubout, and echo. Alternatively, the console task may be connected to a built-in screen and keyboard.

You can have multiple invocations of any of these tasks. For example, if you have three different disks on your machine, then you have three invocations of the file system handler task, which all share the same code.

1.2.2 CLI

The second type of task within the Tripos system is the CLI task. CLI stands for the Command Line Interpreter. This task communicates with the user: it asks for the names of programs to be run, loads them from disk, and executes them. As with other tasks, you may create multiple invocations of the CLI task with the NEWCLI command.

1.2.3 Debug

The Debug task is not always in the system, but if it is, it is a permanently resident system debugger. The user can connect to this task and obtain debugging information about the system.

1.2.4 User Tasks

User tasks are created by a CLI program and can take any form required. They normally send and receive packets in which information is passed. They may call the standard device handler tasks or they may send packets directly to devices. For example, a user task might be a background print spooler created for a word processing package.

1.3 Packets

This section is complex and need only be read by an advanced programmer who wishes to learn about the fundamental internal workings of the Tripos operating system. It is not essential, therefore, to read this before starting to program under Tripos.

Communication between tasks and between task and devices is achieved by sending special messages, called packets. (Other systems may call packets 'messages', but the principle is the same.) To send a packet to another task, you allocate a piece of memory and initialize certain fields within it. All packets in Tripos have standard structure as follows. A packet must be long word aligned, and has the following general structure.

Value	Function	Description
BPTR	pktLink	Link to next packet on chain
LONG	PktDest	Destination
LONG	PktType	Packet type
LONG	Res1	First result field
LONG	Res2	Second result field
LONG	Arg1	Argument; depends on packet type
LONG	Arg2	Argument; depends on packet type
	...	
LONG	ArgN	Argument; depends on packet type

The format of a specific packet depends on its type; but in all cases, it contains a link field which should be set to -1, the identity of the destination, the packet type and two result fields. When Tripos sends a packet, the destination field is overwritten with the task identifier of the sender so that the packet can be returned.

If the destination is a Tripos device handler task or device driver, then the packet contains a value in the PktType field which indicates an action to be performed, such as reading some data. The argument fields contain specific information such as the address and size of the buffer where the characters go.

1.3.1 QPkt and TaskWait

Packets are moved from one task to another, or from a task to a device driver, by the system primitive routine called QPkt. This transmits the packet to the destination by simply copying a pointer to the space. The packet itself is not physically moved. It is clear that, having sent a packet, a routine really expects to get it back because it owns the piece of memory. Although it is not strictly required, it is usual that whenever a packet is sent, the sender expects the packet to return, normally with a reply in it.

A task can pick up any packets that are waiting for it by calling a routine called TaskWait. The TaskWait routine looks to see if there is a packet waiting for that task. There can be many packets waiting for a single

task. These packets are automatically queued by the system. The TaskWait routine gets the next packet from this queue. If there isn't a packet waiting, TaskWait puts the task to 'sleep.' In this state the task does not run; it just waits for something to happen and gives another task a chance to run.

While a packet is 'away' at another task or device the memory which represents the packet must not be altered by the sender. After a call to QPkt and before a call to TaskWait has returned the packet is under the control of the recipient, and the sender should only alter (or deallocate) the message space once the packet has been sent back.

1.3.2 Device Handlers

As described earlier, each device under Tripos consists of two parts: a device handler and a device driver. A call such as Read is implemented by sending a packet to the device handler and this mechanism is described in more detail here.

First of all a CLI task calls the function Read. Then the Read function queues a packet to the device handler task that handles that particular device (for example, if the I/O refers to a disk file, a packet is sent to the file system task). The message has a standard form that can be roughly translated as: "Read these bytes into this buffer which is so long." The CLI task then immediately calls TaskWait to wait for the reply to the message. In the meantime, the handler task, which has been waiting for some work, wakes up with a message that says something like "These bytes are required." The handler task does the work that needs to be done and it either fulfils the request or it does not; it returns the packet to the CLI task anyway to say whether it worked or not. This scheme means that although the majority of programs perform synchronous I/O (in other words, they call Read which goes QPkt followed immediately by TaskWait()), it is not a requirement and it is perfectly possible to handle asynchronous I/O. In this case the application program simply queues a packet containing the read request to start the Read function and then carries on doing something else. At a later point the application must call TaskWait to pick up the reply. The actual packet types which must be sent are described in more detail in the next section.

1.3.3 Device Handler Packet Types

Tripos supports the following packet types to be sent to device handlers. Not all types are valid to all handlers, for example a rename request is only valid to handlers supporting a filing system. For each packet type the arguments and results are described. The actual decimal code for each type appears next to the symbolic name. In all cases, the Res2 field contains additional information concerning an error (indicated by a zero value for Res1 in most cases). To obtain this additional information, you can call IoErr when making a standard Tripos call.

```
Open Old File
Type        LONG   Action.FindInput (1005)

Arg1        BPTR   FileHandle
Arg2        BPTR   Lock
Arg3        BSTR   Name

Res1        LONG   Boolean
```

Attempts to open an existing file for input or output (see the function Open in Chapter 2, "Calling Tripos," of the *Tripos Programmer's Reference Manual* for further details on opening files for I/O). The file is opened with a shared (read) lock so that other tasks may read from the same file.

To obtain the value of lock, you call DeviceProc to obtain the handler taskid (processid) and then IoErr which returns the lock. Alternatively the lock and taskid can be obtained directly from the DevInfo structure. Note that the lock refers to the directory owning the file, not to the file itself.

The caller must allocate and initialize FileHandle. This is done by clearing all fields to zero except for the CharPos and BufEnd fields which should be set to -1. The ProcessID field within the FileHandle must be set to the processid of the handler task.

The result is zero if the call failed, in which case the Res2 field provides more information on the failure and the FileHandle should be released.

Open Old File Exclusive

```
Type      LONG  Action.FindUpdate (1004)

Arg1      BPTR  FileHandle
Arg2      BPTR  Lock
Arg3      BSTR  Name

Res1      LONG  Boolean
```

Arguments as for previous entry, except that the file is locked with an exclusive (write) lock. This is useful for an application such as a database, where the file must not be altered by any other task.

Open New File

```
Type      LONG  Action.FindOutput (1006)

Arg1      BPTR  FileHandle
Arg2      BPTR  Lock
Arg3      BSTR  Name

Res1      LONG  Boolean
```

Arguments as for previous entry, except that the file is created if it did not exist.

Read

```
Type      LONG  Action.Read (82)

Arg1      BPTR  FileHandle Arg1
Arg2      APTR  Buffer
Arg3      LONG  Length

Res1      LONG  Actual Length
```

To read from a file handle, the task id is extracted from the ProcessID field of the file handle, and the Arg1 field from the handle is placed in the Arg1 field of the packet. The buffer address and length are then placed in the other two argument fields. The result indicates the number of characters read; see the function Read for details. An error is indicated by returning -1 whereupon the Res2 field contains more information.

Write

```
Type      LONG  Action.Write (87)

Arg1      BPTR  FileHandle Arg1
Arg2      APTR  Buffer
Arg3      LONG  Length

Res1      LONG  Actual Length
```

The arguments are the same as those for Read. See the Write function for details of the result field.

Close

```
Type      LONG  Action.End (1007)

Arg1      BPTR  FileHandle Arg1

Res1      LONG  TRUE
```

You use this packet type to close an open file handle. The task id of the handler is obtained from the file handle. The function normally returns TRUE. After a file handle has been closed, the space associated with it should be returned to the free pool.

Seek

```
Type      LONG  Action.Seek (1008)

Arg1      BPTR  FileHandle Arg1
Arg2      LONG  Position
Arg3      LONG  Mode

Res1      LONG  OldPosition
```

This packet type corresponds to the SEEK call. It returns the old position, or -1 if an error occurs. The task id is obtained from the file handle.

WaitChar

```
Type        LONG   Action.WaitChar (20)

Arg1        LONG   Timeout

Res1        LONG   Boolean
```

This packet type implements the WaitForChar function. You must send the packet to a console handler task, with the timeout required in Arg1. The packet returns when either a character is waiting to be read, or when the timeout expires. If the result is TRUE, then at least one character may be obtained by a subsequent READ.

ExamineObject

```
Type        LONG   Action.ExamineObject (23)

Arg1        BPTR   Lock
Arg2        BPTR   FileInfoBlock

Res1        LONG   Boolean
```

This packet type implements the Examine function. It extracts the task id of the handler from the ProcessID field of the lock. If the lock is zero, then it uses the default file handler, which is kept in the FiHand field of the task. The result is zero if it fails, with more information in Res2. The FileInfoBlock returns with the name and comment fields as BSTRs.

ExamineNext

```
Type        LONG   Action.ExamineNext (24)

Arg1        BPTR   Lock
Arg2        BPTR   FileInfoBlock

Res1        LONG   Boolean
```

This call implements the ExNext function, and the arguments are similar to those for Examine above. Note that the BSTR representing the filename must not be disturbed between calls of ExamineObject and different calls to ExamineNext, as it uses the name as a place saver within the directory being examined.

DiskInfo

```
Type      LONG  Action.DiskInfo (25)

Arg1      BPTR  InfoData

Res1      LONG  TRUE
```

This implements the Info function. A suitable lock on the device would normally obtain the task id for the handler. This packet can also be sent to a console handler task, in which case the Volume field in the InfoData contains the window pointer for the window opened on your behalf by the console handler.

Parent

```
Type      LONG  Action.Parent (29)

Arg1      BPTR  Lock

Res1      LONG  ParentLock
```

This packet returns a lock representing the parent of the specified lock, as provided by the ParentDir function call. Again it must obtain the task id of the handler from the lock, or from the Fihand field of the current task if the lock is zero.

DeleteObject

```
Type      LONG  Action.DeleteObject (16)

Arg1      BPTR  Lock
Arg2      BSTR  Name

Res1      LONG  Boolean
```

This packet type implements the Delete function. It must obtain the lock from a call to IoErr() immediately following a successful call to DeviceProc which returns the task id. The lock actually refers to the directory owning the object to be deleted, as in the the Open New and Open Old requests.

CreateDir

```
Type        LONG   Action.CreateDir (22)

Arg1        BPTR   Lock
Arg2        BSTR   Name

Res1        BPTR   Lock
```

This packet type implements the CreateDir function. Arguments are the same as for DeleteObject. The result is zero or a lock representing the new directory.

LocateObject

```
Type        LONG   Action.LocateObject (8)

Arg1        BPTR   Lock
Arg2        BSTR   Name
Arg3        LONG   Mode

Res1        BPTR   Lock
```

This implements the Lock function and returns the lock or zero. Arguments as for CreateDir with the addition of the Mode as arg3. Mode refers to the type of lock, shared or exclusive.

CopyDir

```
Type        LONG   Action.CopyDir (19)

Arg1        BPTR   Lock

Res1        BPTR   Lock
```

This implements the DupLock function. If the lock requiring duplication is zero, then the duplicate is zero. Otherwise, the task id is extracted from the lock and this packet type sent. The result is the new lock or zero if an error was detected.

FreeLock

```
Type      LONG  Action.FreeLock (15)

Arg1      BPTR  Lock

Res1      LONG  Boolean
```

This call implements the UnLock function. It obtains the task id from the lock. Note that freeing the zero lock takes no action.

SetProtect

```
Type      LONG  Action.SetProtect (21)

Arg1      Not used
Arg2      BPTR  Lock
Arg3      BSTR  Name
Arg4      LONG  Mask

Res1      LONG  Boolean
```

This implements the SetProtection function. The lock is a lock on the owning directory obtained from DeviceProc as described for DeleteObject above. The least significant four bits of 'Mask' represent Read, Write, Execute, and Delete in that order. Delete is bit zero.

SetComment

```
Type      LONG  Action.SetComment (28)

Arg1      Not used
Arg2      BPTR  Lock
Arg3      BSTR  Name
Arg4      BSTR  Comment

Res1      LONG  Boolean
```

This implements the SetComment function. Arguments as for SetProtect above, except that arg4 is a BSTR representing the comment.

RenameObject

```
Type        LONG   Action.RenameObject (17)

Arg1        BPTR   FromLock
Arg2        BPTR   FromName
Arg3        BPTR   ToLock
Arg4        BPTR   ToName

Res1        LONG   Boolean
```

This implements the Rename function. It must contain an owning directory lock and a name for both the source and the destination. The owning directories are obtained from DeviceProc as mentioned under the entry for DeleteObject.

Inhibit

```
Type        LONG   Action.Inhibit (31)

Arg1        LONG   Boolean

Res1        LONG   Boolean
```

This packet type implements a filing system operation that is not available as a Tripos call. The packet contains a Boolean value indicating whether the filing system is to be stopped from attempting to verify any new disks placed into the drive handled by that handler task. If the Boolean is true, then you may swap disks without the filesystem task attempting to verify the disk. While disk change events are inhibited, the disk type is marked as "Not a DOS disk" so that other tasks are prevented from looking at the disk.

If the Boolean is false, then the filesystem reverts to normal after having verified the current disk in the drive.

This request is useful if you wish to write a program such as DISKCOPY where there is much swapping of disks that may have a half completed structure. If you use this packet request then you can avoid having error messages from the disk validator while it attempts to scan a half completed disk.

RenameDisk

```
Type      LONG  Action.RenameDisk (9)

Arg1      BPTR  NewName

Res1      BPTR  Boolean
```

Again, this implements an operation not normally available through a function call. The single argument indicates the new name required for the disk currently mounted in the drive handled by the file system task where the packet is sent. The volume name is altered both in memory and on the disk.

Flush

```
Type      LONG  Action.Flush (27)
```

This packet type implements another operation not normally available through a function call. It tells the filing system to write out any cached blocks and to turn off the disk light. This call is useful in an application program such as a database where the program wishes to make sure that the disk really has been updated.

GetParameters

```
Type      LONG      Action.GetParameters ( 991 )

Arg1      BPTR      Parameter buffer
```

The buffer structure of the argument is as follows:

Value	Function	Description
LONG	Width	The terminal width
BOOL	Tab	Software tab expansion flag (TRUE = ON, FALSE = OFF)
LONG	Process	Process switching code or 0 if turned off
LONG	PageLength	Height of the terminal screen in lines
LONG	TabStop	Tab expansion size if expansion flag is ON
BOOL	AutoNL	Automatic \n at end of line flag
BOOL	IFC	Input flow control enabling flag

This packet type will return from a console handler the currently set values of the parameters as shown above.

SetParameters

Type	LONG	Action.SetParameters (990)
Arg1	BPTR	Parameter buffer

The buffer structure of the argument is as shown above under GetParameters

This packet type will set the parameters given in the buffer as the current working parameters in a console handler.

Act.SC.Mode

Type	LONG	Act.SC.Mode (994)
Arg1	BOOL	Single character mode flag

The single argument indicates to a console handler what mode all subsequent read or write requests will be performed in for the issuing task. If the argument is TRUE, the console handler is placed into single character mode. In single character mode all output requests are sent to

the screen untranslated by any options that may be in effect at the time. This also means that read requests are returned as soon as the requested buffer is full (that is, not when the user presses RETURN).

1.3.4 Device Drivers

Packets are also used to communicate between tasks and devices. The device handler task needs to communicate with its associated device driver and it does this by sending packets. In much the same way as an application program creates a packet and sends it to a device handler, the device handler receives packets from a program, creates further packets, and sends these to the device driver. The device driver never creates any packets; it receives packets from the handler and sets up an I/O request to the hardware to perform this action. For example, when a disk device driver receives a request to read data from the disk, it sets up the hardware to start doing the read. It does not send the packet back because the read has not finished yet. When the device driver has set up the request some other task within the system is free to execute. Eventually, the device driver fields an interrupt. Hopefully it is the interrupt saying that the read has completed; in which case the device driver takes the packet in question, puts the return code into it, and sends the packet back. As far as the device handler task is concerned, it sends a packet to the device driver and gets the packet when the job is done.

1.3.5 Device Driver Packet Types

Packets sent to a device have the following general structure.

Field	Name	Description
0	Link	Link to another packet
4	Taskid	Task identifier of sender
8	Type	Command type
12	Res1	Primary result
16	Res2	Secondary result
20	Unit	Unit number
24	Buffer	Pointer to buffer area
28	Size	Size of buffer area
32	Offset	Byte offset

The standard command types are Read, Write, and Reset. All devices support at least these three types. The value of byte offset is usually only used by devices such as disks, and specifies the offset in bytes where the read or write is to start. The value of Res1 is normally the actual number of bytes read or written. If it is zero then an error has occurred, and more information is available in Res2.

Specific Device Commands and Arguments

Serial line

```
Command: GetParam
Buffer:  Baud Rate Long
                   300, 9600, etc
         Parity    Long
                   0=none,1=odd,2=even,3=space,4=mark
         Data bits Long
                   5, 6, 7, or 8
         Stop bits Long
                   0=1 bit, 1=1.5 bits, 2=2 bits
```

Used to get the buffer.

```
Command: SetParam
Buffer:  Baud Rate Long
                   300, 9600, etc
         Parity    Long
                   0=none,1=odd,2=even,3=space,4=mark
         Data bits Long
                   5, 6, 7 or 8
         Stop bits Long
                   0=1 bit, 1=1.5 bits, 2=2 bits
```

Used to set the parameters for a serial device.

Disk

```
Command: FormatTrack
Buffer:  not used
```

Used to format an entire track, destroying any previous information.

```
Command: Status
Buffer:  not used
Resl:    0 if disk in drive, 1 if disk read only,
         2 if disk if disk not ready
```

Determines whether there is a disk in the drive or not. Ideally, this request is sent by the file system and only returns when a disk is inserted or removed. However, as some hardware may not be capable of supporting this, the packet may be returned at once with the current status of the disk. The packet is sent again every 3 seconds or so, and each time the device must determine whether a disk is in the drive or not.

```
Command: MotorOff
Buffer:  not used
```

Used to turn the motor and the associated drive select light on a floppy disk off. It may be ignored for a hard disk. When the select light is off, you may remove the disk from the drive.

Chapter 2: Calling the Kernel

This chapter describes the functions provided by the Tripos kernel. To help you, it provides the following: an explanation of the syntax, a full description of each function, and a quick reference card of the available functions.

Table of Contents

2.1 Syntax

The syntax used in this chapter shows the C function call for each Tripos function and the corresponding register you use when you program in assembler.

1. Register values

The letter/number combinations (D0...Dn) represent registers. The text to the left of an equals sign represents the result of a function. A register (that is, D0) appearing under such text indicates the register value of the result. Text to the right of an equals sign represents a function and its arguments, where the text enclosed in parentheses is a list of the arguments. A register (for example, D2) appearing under an argument indicates the register value of that argument.

Note that not all functions return a result, and not all functions take an argument.

2. Case

The letter case (that is, lower or upper case) IS significant. For example, you must enter the word 'GetMem' with the first letter of each component word in upper case.

3. Boolean returns

Non-zero (TRUE or SUCCESS), zero (FALSE or FAILURE).

4. Values

All values are longwords (that is, 4 byte values or 32 bits).

5. Code, Format, Argument, and Result

'Code:' introduces the function code. 'Argument:' and 'Result:' provide further details on the syntax used after 'Format:'. Result describes what is returned by the function (that is, the left of the equal sign). Argument describes what the function expects to work on (that is, the list in parentheses). Figure 2-A should help explain the syntax.

```
Format of function      result = Function( argument )
                        Register           Register

Example                 vector := GetMem( size )
                        D0                 D1
```

Figure 2-A: Format of Functions and Registers

When calling kernel functions (routines) in C, use the call sequence as shown above. When calling in Assembler, place the arguments in the registers as shown and place the function code (the number listed after 'Code:') in D0. Then perform TRAP #0. Results are returned in D0 and D1; all other registers are preserved.

If a function fails, it returns the value 0 to D0 and outputs the relevant error code to D1.

2.2 Kernel Functions

This reference section describes the functions provided by the Tripos kernel. Each function is arranged alphabetically under the following headings: Memory Management, Task Management, Device Management, Message Passing, and Miscellaneous. These headings indicate the action of the functions they cover. Under each function name, there is a brief description of the function's purpose, a specification of the format and the register values, a fuller description of the function, and an explanation of the syntax of the arguments and result.

Memory Management

FreeMem

Purpose: To free memory previously acquired by GetMem.

Code: 2

Form:
```
FreeMem( vector )
          D1
```

Argument: vector = address of a block of memory

Description:

A pointer to a vector allocated by GetMem is passed as argument. An attempt to free a vector which does not appear to have been allocated by GetMem results in the aborting of the issuing task. A call of Freemem(0) is permissable and has no effect.

See also: *GetMem*

GetMem

Purpose: To allocate memory from the system.

Code: 1

Form:
```
vector := GetMem( size )
 D0               D1
```

Argument: size = size in bytes

Result: vector = address of space allocated

Description:

The size of the required vector is passed as argument. The result is zero if the call failed; otherwise it is a pointer to an area of memory with bytes numbered from 0 to size -1.

It is possible that this routine might detect the corruption of free memory, caused, for example, by a previous program. In this case a system abort occurs.

See also: *FreeMem*

Task Management

AddTask

Purpose: To add a new task.

Code: 3

Form:

```
taskid =
  D0
AddTask(segvec, stacksize, pri, name)
          D1       D2       D3   D4
```

Argument:
segvec - pointer to a segment vector
stacksize - integer
pri - integer
name - string

Result: taskid - task identifier

Description:

AddTask creates a task with the name 'name'. That is to say, AddTask allocates a task control structure from the free memory area and then initializes it.

AddTask takes a pointer to a segment vector as the argument 'segvec'. This points to the sections of code that you intend to run as a new task, and includes the shared system library segment as well as the segment containing the code for the new task.

'stacksize' represents the size of the root stack in bytes when AddTask activates the task. 'pri' specifies the required priority of the new task. The result is the task identifier of the new task, or zero if the routine failed.

The argument 'name' specifies the task name.

A zero return code implies an error of some kind.

Note: You would normally use CreateProc instead.

See also: *ChangePri, RemTask*

ChangePri

Purpose: To change the priority of a task.

Code: 35

Form:
```
res := ChangePri( taskid, newpriority )
 D0                 D1       D2
```

Argument: taskid = task identifier
newpriority = new priority

Description:

The routine is passed the task number of the task which is to have its priority changed. The priority that the specified task is to adopt is passed as newpriority; this is a positive number. Lowest priority tasks have small numbers; for example, any task created by the RUN command has a priority around 500 while a CLI has a priority around 1000. The result is non-zero if it worked. Calling this routine may cause a change of current task.

See also: *AddTask, RemTask*

Forbid

Purpose: To disallow task rescheduling.

Code: 6

Form: `Forbid()`

Description:

A task which issues Forbid stops any other task from running. It is as though every other task in the system had been held (see also Permit and Hold).

Once a task has issued Forbid, it cannot, for example, perform any I/O as this requires another task to run. The main use of Forbid and Permit is to enclose a piece of code which manipulates a shared datastructure. You can ensure that two programs do not alter the same datastructure at the same time (with disastrous consequences) by surrounding the sensitive code with a Forbid/Permit pair.

See also: *Permit, Hold*

Hold

Purpose: To place a task into HELD state.

Code: 9

Form:
```
result := Hold( taskid )
 D0                 D1
```

Argument: taskid = task identifier

Description:

Hold places the specified task into held state. The task will perform no work until it is released. The result is non-zero if the routine worked.

See also: *Release*

Permit

Purpose: To allow task rescheduling.

Code: 5

Form: `Permit()`

Description:

Permit undoes the effect of Forbid. You may only use Permit after using Forbid.

See also: *Forbid*

Release

Purpose: To place a task in an unHELD state.

Code: 10

Form:
```
result := Release( taskid )
 D0                    D1
```

Argument: taskid = task identifier

Description:

Release reverses the effect of Hold. That is to say, it releases a held task and allows it to run. The result is non-zero if the call worked.

See also: *Hold*

RemTask

Purpose: To remove a task from the system.

Code: 4

Form:
```
res := RemTask( taskid )
 D0                 D1
```

Argument: taskid = task identifier

Description:

The task specified by the taskid is deleted if possible. It is only possible to delete tasks which have an empty work queue (that is, which have no packets waiting to be extracted via TaskWait). In addition, only the current task or a task which is 'dead' may be deleted.

The result is non-zero if the deletion worked - the returned value will not, of course, be available if the current task is being deleted. A value of zero indicates an error.

See also: *AddTask, EndTask*

SetFlags

Purpose: To set attention flags.

Code: 36

Form:
```
result := SetFlags( taskid,  mask )
 D0                 D1       D2
```

Argument: taskid = task identifier of task in which to set flags
mask = bit mask of flags set

Description:

The id of the task whose flags are to be set is passed, along with a mask indicating which flags are to be set. A binary 1 indicates selection, 0 otherwise. Four flags are available, corresponding to the action of CTRL-C to CTRL-F typed at the task. The result is non-zero if the flags have been set as requested.

See also: *TestFlags*

SuperMode

Purpose: To enter supervisor mode.

Code: 7

Form: `SuperMode()`

Description:

The current task is set into supervisor state.

See also: *UserMode*

TestFlags

Purpose: To test the task's flag word.

Code: 17

Form:
```
result := TestFlags( mask )
 D0                   D1
```

Argument: mask = bit mask of flags to test

Result: result = mask of flags which are set

Description:

The argument 'mask' indicates the selection of flags to be tested. A binary 1 indicates selection, 0 otherwise. There are four flags available, corresponding to CTRL-C to CTRL-F typed at the terminal, or set via SetFlags.

The result is TRUE if at least one of the selected flags is set. Register D1 then holds the result of ANDing the contents of the flagword (prior to testing), with the specified mask. The result is FALSE if none of the selected flags is set. All selected flags are cleared.

It is normal for all programs written under Tripos to call TestFlags(1) at suitable points to determine if the user has pressed CTRL-C. If the routine returns TRUE then the program should normally terminate after issuing an appropriate message.

See also: *SetFlags*

UserMode

Purpose: To exit to user mode.

Code: 8

Form: `UserMode()`

Description:

UserMode lets you put the task back into user mode after using SuperMode.

See also: *SuperMode*

Device Management

AddDevice

Purpose: To add a new device to the system.

Code: 19

Form:
```
deviceid := AddDevice( address )
 D0                    D1
```

Argument: address = DCB address

Result: deviceid = device identifier

Description:

AddDevice initializes the device associated with the Device Control Block (DCB). It initializes the interrupt vectors specified inthe DCB, and calls the device Open entry point. The deviceid is zero if AddDevice fails; otherwise it is the device identifier.

See also: *RemDevice*

RemDevice

Purpose: To remove a device from the system.

Code: 20

Form:
```
RemDevice( deviceid )
             D1
```

Argument: deviceid = device identifier

Description:

The device identifier from a previous call of AddDevice is the deviceid. RemDevice calls the device Close entry point, and resets the interrupt vector associated with the device.

See also: *AddDevice*

Message Passing

DQPkt

Purpose: To reclaim a packet.

Code: 43

Form:

```
bool := DQPkt( packet, id )
                  D1      D2
```

Argument: packet = packet address
id = task/device identifier

Result: bool = boolean return

Description:

The routine is passed the id of a task or device whose work queue is searched first for the packet specified as 'packet'. The result is True if the packet is reclaimed; otherwise it is False (for example, if the packet is not on the queue).

See also: *QPkt, TaskWait, TestWkQ*

FindTask

Purpose: To return the identity of the current task.

Code: 21

Form: `tcb := FindTask()`

Result: tcb = current task control block

Description:

The routine returns the current task control block, which is used as a parameter to many of the kernel calls.

See also: *QPkt, TestWkQ*

QPkt

Purpose: To send a message.

Code: 12

Form:
```
result := QPkt( packet, senderid )
 D0              D1       D2
```

Argument: packet = address of packet
senderid = identifier of sender

Description:

This routine is central to the action of Tripos. The packet is merely a pointer to a block of memory with certain fields used to hold specific items of information. The format of packets to be sent to specific device handlers and tasks, as well as a general description of packets, are provided in Chapter 1 of this manual.

All fields in a packet are longwords, and the packet structure must be longword aligned. This can be achieved by either using space obtained by a call to GetMem, or by ensuring that only longword values are declared on the stack from which the packet has been obtained.

The first element is used by the system as a link field, and must be set, before the call to QPkt, to -1. Note that when a packet is returned by TaskWait the value -1 is replaced in this field.

The second element must indicate the destination. This is positive for a task and negative for a device. The timer device always has the identity -1. This field will be replaced by the value given as senderid, so that the packet will be returned to the sender. This value may refer to some other task if required but this will cause the packet to be returned to another task which must be expecting this. Normally the value specified as senderid will be the identity of the current task or device. For a task, this is obtained from the id field of the current TCB. The current TCB is returned from a FindTask kernel call.

The third element in a packet is used to hold a type code while the next two elements are used as result fields and the rest are used as argument fields.

The result will be non-zero if the packet was sent successfully, and the first element of the packet will be set to a value other than -1. The second element will be set to the issuing task's id. Calling this routine may cause a change of current task. An error occurs if the first element is not equal to -1 or if the destination is invalid.

See also: *DQPkt, FindTask, TaskWait, TestWkQ*

TaskWait

Purpose: To wait for the next packet to arrive at the task.

Code: 41

Form:
```
packet := TaskWait()
 D0
```

Result: packet = address of next packet arriving at this task

Description:

The routine checks to see if there is a packet waiting for this task; if so, the packet is removed from the work queue and returned as the result. If the task queue is empty, then the routine waits indefinitely for a packet to arrive. When it does, the address of the packet is returned as the result. The packet link field is cleared to -1, and the destination field of the packet will have been set to the identity of the packet sender. Thus the packet returned from TaskWait is ready to be returned to the sender via the QPkt call.

See also: *DQPkt, QPkt, TestWkQ*

TestWkQ

Purpose: To test if there is anything on the work queue.

Code: 14

Form:
```
bool := TestWkQ(tcb)
                 D1
```

Argument: tcb = task control block

Result: bool = boolean return

Description:

The result is TRUE if there is a packet waiting on the work queue of the task specified as the argument; otherwise it is FALSE. This is useful when a program wishes to detect whether an event has happened, rather than waiting indefinitely for the event to occur.

See also: *DQPkt, FindTask, QPkt, TaskWait*

Miscellaneous

FindDOS

Purpose: To return the DOS library base pointer.

Code: 18

Form:
```
address := FindDOS( )
 D0
```

Result: address = address of DOS library

Description:

This function returns the base library pointer, which is used to call any of the functions listed in the next chapter.

RootStruct

Purpose: To return a pointer to the rootnode.

Code: 22

Form:
```
address = RootStruct( )
 D0
```

Result: address = address of the rootnode

Description:

This function returns a pointer in the rootnode. For further details of this structure, see Chapter 3, "Tripos Data Structure," of the *Tripos Technical Reference Manual*.

Quick Reference Card

Memory Management:

FreeMem	GetMem

Task Management:

AddTask	Permit	Supermode
ChangePri	Release	TestFlags
Forbid	RemTask	UserMode
Hold	SetFlags	

Device Management:

AddDevice	RemDevice

Message Passing:

DQPkt	QPkt	TestWkQ
FindTask	TaskWait	

Miscellaneous:

FindDOS	RootStruct

AddDevice	To add a new device to the system.
AddTask	To add a new task.
ChangePri	To change the priority of a task.
DQPkt	To reclaim a packet.
FindDOS	To return the DOS library base pointer.
FindTask	To return current task identity.
Forbid	To disallow task rescheduling.
FreeMem	To free memory previously acquired by GetMem.
GetMem	To allocate memory from the system.
Hold	To place a task into HELD state.
Permit	To allow task rescheduling.
QPkt	To send a message.
Release	To place a task in an unHELD state.
RemDevice	To remove a device from the system.
RemTask	To remove a task from the system.
RootStruct	To return a pointer to the rootnode.
SetFlags	To set attention flags.
SuperMode	To enter supervisor mode.
TaskWait	To wait for the next packet to arrive at the task.

TestFlags	To test the task's flag word.
TestWkQ	To test if there is anything on the work queue.
UserMode	To exit to user mode.

Chapter 3: Calling the DOS

This chapter describes the functions provided by the Tripos Disk Operating System (DOS). To help you, it provides the following: an explanation of the syntax, a full description of each function, and a quick reference card of the available functions.

Table of Contents

3.1 Syntax

The syntax used in this chapter shows the C function call for each Tripos function and the corresponding register you use when you program in assembler.

1. Register values

The letter/number combinations (D0...Dn) represent registers. The text to the left of an equals sign represents the result of a function. A register (that is, D0) appearing under such text indicates the register value of the result. Text to the right of an equals sign represents a function and its arguments, where the text enclosed in parentheses is a list of the arguments. A register (for example, D2) appearing under an argument indicates the register value of that argument.

Note that not all functions return a result.

2. Case

The letter case (that is, lower or upper case) IS significant. For example, you must enter the word 'FileInfoBlock' with the first letter of each component word in upper case.

3. Boolean returns

-1 (TRUE or SUCCESS), 0 (FALSE or FAILURE).

4. Values

All values are longwords (that is, 4 byte values or 32 bits). Values referred to as "string" are 32 bit pointers to NULL terminated series of characters.

5. Format, Argument and Result

Look at 'Argument:' and 'Result:' for further details on the syntax used after 'Format:'. Result describes what is returned by the function (that is, the left of the equal sign). Argument describes what the function expects to work on (that is, the list in parentheses). Figure 3-A should help explain the syntax.

```
Format of function      result = Function( argument )
                        Register           Register

Example                 lock = CreateDir( name )
                        D0                D1
```

Figure 3-A: Format of Functions and Registers

3.2 Tripos Functions

This reference section describes the functions provided by the Tripos Disk Operating System. Each function is arranged alphabetically under the following headings: File Handling, Task Handling, and Loading Code. These headings indicate the action of the functions they cover. Under each function name, there is a brief description of the function's purpose, a specification of the format and the register values, a fuller description of the function, and an explanation of the syntax of the arguments and result. To use any of these functions, you must link with the DOS library. The DOS library interface routines will be supplied in the run-time support library for the language you are using (for example, C = CLIB).

File Handling

Close

Purpose: To close a file for input or output.

Form:
```
Close( file )
       D1
```

Argument: file - file handle

Description:

The file handle 'file' indicates the file that Close should close. You obtain this file handle as a result of a call to Open. You must remember to close explicitly all the files you open in a program. However, you should not close inherited file handles opened elsewhere.

See also: *Open*

CreateDir

Purpose: To create a new directory.

Form:
```
lock = CreateDir( name )
D0                D1
```

Argument: name - string

Result: lock - pointer to a lock

Description:

CreateDir creates a new directory with the name you specified, if possible. It returns an error if it fails. Remember that Tripos can only create directories on devices which support them, for example, disks.

A return of zero means that Tripos has found an error (such as: disk write protected), you should then call IoErr(); otherwise, CreateDir returns a shared read lock on the new directory.

CurrentDir

Purpose: To make a directory associated with a lock the current working directory.

Form:
```
oldLock = CurrentDir( lock )
D0                    D1
```

Argument: lock - pointer to a lock

Result: oldLock - pointer to a lock

Description:

CurrentDir makes current a directory associated with a lock. It returns the old current directory lock.

A value of zero is a valid result here and indicates that the current directory is the root of the initial start-up disk.

See also: *Lock*

DeleteFile

Purpose: To delete a file or directory.

Form:
```
success = DeleteFile( name )
D0                    D1
```

Argument: name - string

Result: success - boolean

Description:

DeleteFile attempts to delete the file or directory 'name'. It returns an error if the deletion fails. Note that you must delete all the files and any directories within a directory before you can delete the directory itself.

DupLock

Purpose: To duplicate a lock.

Form:
```
newLock = DupLock( lock )
D0                 D1
```

Argument: lock - pointer to a lock

Result: newLock - pointer to a lock

Description:

DupLock takes a shared filing system read lock and returns another shared read lock to the same object. It is impossible to create a copy of a write lock. (For more information on locks, see under Lock.)

Examine

Purpose: To examine a directory or file associated with a lock.

Form:
```
success = Examine( lock, FileInfoBlock )
 D0                D1     D2
```

Argument: lock - pointer to a lock
FileInfoBlock - pointer to a file info block

Result: success - boolean

Description:

Examine fills in information in the FileInfoBlock concerning the file or directory associated with the lock. This information includes the name, size, creation date, and whether it is a file or directory.

Note: FileInfoBlock must be longword aligned. You can ensure this in the language C if you use GetMem. (See Chapter 2, "Calling the Kernel," for further details on GetMem.)

Examine gives a return code of zero if it fails.

ExNext

Purpose: To examine the next entry in a directory.

Form:
```
success = ExNext( lock, FileInfoBlock )
 D0                D1     D2
```

Argument: lock - pointer to a lock
FileInfoBlock - pointer to a file info block

Result: success - boolean

Description:

This routine is passed a lock, usually associated with a directory, and a FileInfoBlock filled in by a previous call to Examine. The FileInfoBlock contains information concerning the first file or directory stored in the directory associated with the lock. ExNext also modifies the FileInfoBlock so that subsequent calls return information about each following entry in the directory.

ExNext gives a return code of zero if it fails for some reason. One reason for failure is reaching the last entry in the directory. However, IoErr() holds a code that may give more information on the exact cause of a failure. When ExNext finishes after the last entry, it returns ERROR_NO_MORE_ENTRIES

So, follow these steps to examine a directory:

1) Use Examine to get a FileInfoBlock about the directory you wish to examine.

2) Pass ExNext the lock related to the directory and the FileInfoBlock filled in by the previous call to Examine.

3) Keep calling ExNext until it fails with the error code held in IoErr() equal to ERROR_NO_MORE_ENTRIES.

4) Note that if you don't know what you are examining, inspect the type field of the FileInfoBlock returned from Examine to find out whether it is a file or a directory which is worth calling ExNext for.

The type field in the FileInfoBlock has two values: if it is negative, then the file system object is a file; if it is positive, then it is a directory.

Info

Purpose: Returns information about the disk.

Form:
```
success = Info( lock, Info_Data )
 D0              D1     D2
```

Argument: lock - pointer to a lock
Info_Data - pointer to an Info_Data structure

Result: success - boolean

Description:

Info finds out information about any disk in use. 'lock' refers to the disk, or any file on the disk. Info returns the Info_Data structure with information about the size of the disk, number of free blocks and any soft errors. Note that Info_Data must be longword aligned.

Input

Form:

```
file = Input()
 D0
```

Result: file - file handle

Description:

To identify the program's initial input file handle, you use Input.

See also: *Output*

IoErr

Purpose: To return extra information from the system.

Form:

```
error = IoErr()
 D0
```

Result: error - integer

Description:

I/O routines return zero to indicate an error. When an error occurs, call this routine to find out more information. Some routines use IoErr(), for example, DeviceProc, to pass back a secondary result.

IsInteractive

Purpose: To discover whether a file is connected to a virtual terminal or not.

Form:
```
bool = IsInteractive ( file )
 D0                    D1
```

Argument: file - file handle

Result: bool - boolean

Description:

The function IsInteractive gives a boolean return. This indicates whether or not the file associated with the file handle 'file' is connected to a virtual terminal.

Lock

Purpose: To lock a directory or file.

Form:
```
lock  = Lock( name, accessMode )
 D0             D1      D2
```

Argument: name - string
accessMode - integer

Result: lock - pointer to a lock

Description:

Lock returns, if possible, a filing system lock on the file or directory 'name'. If the accessMode is ACCESS_READ, the lock is a shared read lock; if the accessMode is ACCESS_WRITE, then it is an exclusive write lock. If Lock fails (that is, if it cannot obtain a filing system lock on the file or directory) it returns a zero.

Note that the overhead for doing a Lock is less than that for doing an Open, so that, if you want to test to see if a file exists, you should use Lock. Of course, once you've found that it exists, you have to use Open to open it.

Open

Purpose: To open a file for input or output.

Form:
```
file = Open( name, accessMode )
 D0            D1     D2
```

Argument: name - string
accessMode - integer

Result: file - file handle

Description:

Open opens 'name' and returns a file handle. If the accessMode is MODE_OLDFILE (=1005), Open opens an existing file for reading or writing. However, Open creates a new file for writing if the value is MODE_NEWFILE (=1006). The 'name' can be a filename (optionally prefaced by a device name), a simple device such as NIL:, or some other device, such as SER:, or *, meaning the terminal.

Note that OPEN_OLDFILE has a shared lock, whereas OPEN_UPDATE has an exclusive lock.

For further details on these devices see Chapter 1 of the *Introduction to Tripos*. If Open cannot open the file 'name' for some reason, it returns the value zero (0). In this case, a call to the routine IoErr() supplies a secondary error code.

For testing to see if a file exists, see the entry under Lock.

Output

Form:

```
file = Output()
  D0
```

Result: file - file handle

Description:

To identify the program's initial output file handle, you use Output.

See also: *Input*

ParentDir

Purpose: To obtain the parent of a directory or file.

Form:

```
Lock = ParentDir( lock )
 D0                D1
```

Argument: lock - pointer to a lock

Result: lock - pointer to a lock

Description:

This function returns a lock associated with the parent directory of a file or directory. That is, ParentDir takes a lock associated with a file or directory and returns the lock of its parent directory.

Note: The result of ParentDir may be zero (0) for the root of the current filing system.

Read

Purpose: To read bytes of data from a file.

Form:
```
actualLength =
 D0
  Read( file, buffer, length )
          D1     D2       D3
```

Argument: file - file handle
buffer - pointer to buffer
length - integer

Result: actualLength - integer

Description:

You can copy data with a combination of Read and Write. Read reads bytes of information from an opened file (represented here by the argument 'file') into the memory buffer indicated. Read attempts to read as many bytes as fit into the buffer as indicated by the value of length. You should always make sure that the value you give as the length really does represent the size of the buffer. Read may return a result indicating that it read less bytes than you requested, for example, when reading a line of data that you typed at the terminal.

The value returned is the length of the information actually read. That is to say, when 'actualLength' is greater than zero, the value of 'actualLength' is the the number of characters read. A value of zero means that end-of-file has been reached. Errors are indicated by a value of -1. Read from the console returns a value when a return is found or the buffer is full.

A call to Read also modifies or changes the value of IoErr(). IoErr() gives more information about an error (for example, actualLength equals -1) when it is called.

See also: *Write*

Rename

Purpose: To rename a directory or file.

Form:
```
success = Rename( oldName, newName )
 D0                D1       D2
```

Argument: oldName - string
newName - string

Result: success - boolean

Description:

Rename attempts to rename the file or directory specified as 'oldName' with the name 'newName'. If the file or directory 'newName' exists, Rename fails and returns an error.

Both the 'oldName' and the 'newName' can be complex filenames containing a directory specification. In this case, the file will be moved from one directory to another. However, the destination directory must exist before you do this.

Note: It is impossible to rename a file from one volume to another.

Seek

Purpose: To move to a logical position in a file.

Form:
```
oldPosition =
 D0
  Seek( file, position, mode )
          D1     D2         D3
```

Argument:
file - file handle
position - integer
mode - integer

Result: oldPosition - integer

Description:

Seek sets the read/write cursor for the file 'file' to the position 'position'. Both Read and Write use this position as a place to start reading or writing. If all goes well, the result is the previous position in the file. If an error occurs, the result is -1. You can then use IoErr() to find out more information about the error.

'mode' can be OFFSET_BEGINNING (=-1), OFFSET_CURRENT (=0) or OFFSET_END (=1). You use it to specify the relative start position. For example, 20 from current is a position twenty bytes forward from current, -20 from end is 20 bytes before the end of the current file.

To find out the current file position without altering it, you call to Seek specifying an offset of zero from the current position.

To move to the end of a file, Seek to end-of-file offset with zero position. Note that you can append information to a file by moving to the end of a file with Seek and then writing. You cannot Seek beyond the end of a file.

See also: *Read, Write*

SetComment

Purpose: To set a comment.

Form:
```
Success = SetComment( name, comment )
 D0                   D1      D2
```

Argument: name - file name
comment - pointer to a string

Result: success - boolean

Description:

SetComment sets a comment on a file or directory. The comment is a pointer to a null-terminated string of up to 80 characters.

SetProtection

Purpose: To set file, or directory, protection.

Form:
```
Success = SetProtection( name, mask )
 D0                      D1     D2
```

Argument: name - file name
mask - the protection mask required

Result: success - boolean

Description:

SetProtection sets the protection attributes on a file or directory. The lower four bits of the mask are as follows:

bit 3: if 1 then reads not allowed, else reads allowed.
bit 2: if 1 then writes not allowed, else writes allowed.
bit 1: if 1 then execution not allowed, else execution allowed.
bit 0: if 1 then deletion not allowed, else deletion allowed.

Bits 31-4 Reserved.

Only delete is checked for in the current release of Tripos.

UnLock

Purpose: To unlock a directory or file.

Form:
```
UnLock( lock )
        D1
```

Argument: lock - pointer to a lock

Description:

UnLock removes a filing system lock obtained from Lock, DupLock, or CreateDir.

See also: *CreateDir, DupLock, Lock*

WaitForChar

Purpose: To indicate whether characters arrive within a time limit or not.

Form:
```
bool = WaitForChar( file, timeout )
 D0                 D1      D2
```

Argument: file - file handle
timeout - integer

Result: bool - boolean

Description:

If a character is available to be read from the file associated with the handle 'file' within a certain time, indicated by 'timeout', WaitForChar returns -1 (TRUE); otherwise, it returns 0 (FALSE). If a character is available, you can use Read to read it. Note that WaitForChar is only valid when the I/O streams are connected to a virtual terminal device. 'timeout' is specified in system **ticks**.

WaitForChar only works for the console handler in single character mode.

Write

Purpose: To write bytes of data to a file.

Form:
```
returnedLength =
 D0
Write( file, buffer, length )
          D1     D2        D3
```

Argument:
file - file handle
buffer - pointer to buffer
length - integer

Result: returnedLength - integer

Description:

You can copy data with a combination of Read and Write. Write writes bytes of data to the opened file 'file'. 'length' refers to the actual length of data to be transferred; 'buffer' refers to the buffer size.

Write returns a value that indicates the length of information actually written. That is to say, when 'length' is greater than zero, the value of 'length' is the number of characters written. A value of -1 indicates an error. The user of this call must always check for an error return which may, for example, indicate that the disk is full.

Task Handling

CreateProc

Purpose: To create a new task (process).

Form:

```
process =
  D0
CreateProc(name, pri, segment, stackSize)
              D1    D2    D3       D4
```

Argument:
name - string
pri - integer
segment - pointer to a segment
stackSize - integer

Result: process - task (process) identifier

Description:

CreateProc creates a task with the name 'name'. That is to say, CreateProc allocates a task control structure from the free memory area and then initializes it.

CreateProc takes a segment list as the argument 'segment'. This segment list represents the section of code that you intend to run as a new task. CreateProc enters the code at the first segment in the segment list, which should contain suitable initialization code or a jump to such.

'stackSize' represents the size of the root stack in bytes when CreateProc activates the task. 'pri' specifies the required priority of the new task. The result is the task identifier of the new task, or zero if the routine failed.

The argument 'name' specifies the task name.

A zero return code implies an error of some kind.

See also: *LoadSeg, UnLoadSeg*

DateStamp

Purpose: To obtain the date and time in internal format.

Form: `v:= DateStamp( v )`

Argument: v - pointer

Description:

DateStamp takes a vector of three longwords that is set to the current time. The first element in the vector is a count of the number of days. The second element is the number of minutes elapsed in the day. The third is the number of **tick**s elapsed in the current minute. A tick happens 50 times a second. DateStamp ensures that the day, minute, and tick are consistent. All three elements are zero if the date is unset.

Delay

Purpose: To delay a task for a specified time.

Form:
```
Delay( timeout )
         D1
```

Argument: timeout - integer

Description:

The function Delay takes an argument 'timeout'. 'timeout' allows you to specify how long the task should wait in ticks (50 per second).

DeviceProc

Purpose: To return the task (process) identifier of the task handling that I/O.

Form:
```
process = DeviceProc( name )
 D0                    D1
```

Argument: name - string

Result: process - task (process) identifier

Description:

DeviceProc returns the task identifier of the task that handles the device associated with the specified name. If DeviceProc cannot find a task handler, the result is zero. If 'name' refers to a file on a mounted device, then IoErr() returns a pointer to a directory lock.

Exit

Purpose: To exit from a program.

Form:
```
Exit( returnCode )
        D1
```

Argument: returnCode - integer

Description:

Exit acts differently depending on whether you are running a program under a CLI or not. If you run, as a command under a CLI, a program that calls Exit, the command finishes and control reverts to the CLI. Exit then interprets the argument 'returnCode' as the return code from the program.

If you run the program as a distinct task, Exit deletes the task and releases the space associated with the stack and task structure.

Note: The space associated with the segment list is not released.

Loading Code

Execute

Purpose: To execute a CLI command.

Form:
```
Success =
 D0
 Execute( commandString, input, output )
             D1           D2     D3
```

Argument: commandString - string
input - file handle
output - file handle

Result: Success - boolean

Description:

This function takes a string (commandString) that specifies a CLI command and arguments, and attempts to execute it. The CLI string can contain any valid input that you could type directly at a CLI, including input and output indirection using > and <.

The input file handle will normally be zero, and in this case the Execute command will perform whatever was requested in the commandString and then return. If the input file handle is non-zero then after the (possibly null) commandString is performed subsequent input is read from the specified input file handle until end of file is reached.

In most cases the output file handle must be provided, and will be used by the CLI commands as their output stream unless redirection was specified. If the output file handle is set to zero then the current output, specified as *, is used.

The Execute function may also be used to create a new interactive CLI task just like those created with the NEWCLI function. In order to do this you should call Execute with an empty commandString, and pass a file handle relating to an interactive input stream as the input file handle. The output file handle should be set to zero. The CLI will read commands from the input stream, and will use the same stream for output. This new CLI can only be terminated by using the ENDCLI command. For this command to work the program C:RUN must be present in C:.

LoadSeg

Purpose: To load a load module into memory.

Form:
```
segment = LoadSeg( name )
 D0                 D1
```

Argument: name - string

Result: segment - pointer to a segment

Description:

The file 'name' is a load module produced by the linker. LoadSeg takes this and scatter loads the code segments into memory, chaining the segments together on their first words. It terminates the list with a zero link word to indicate the end of the chain.

If an error occurs, LoadSeg unloads any loaded blocks and returns a false (zero) result.

If all goes well (that is, LoadSeg has loaded the module correctly), then Loadseg returns a pointer to the beginning of the list of blocks. Once you have finished with the loaded code, you can unload it with a call to UnLoadSeg. (For using the loaded code, see under CreateProc.)

See also: *CreateProc, UnLoadSeg*

UnLoadSeg

Purpose: To unload a segment previously loaded by LoadSeg.

Form:
```
UnLoadSeg( segment )
             D1
```

Argument: segment - pointer to a segment

Description:

UnLoadSeg unloads the segment identifier that was returned by LoadSeg. 'segment' may be zero.

Miscellaneous

VDU

Purpose: To set up full-screen support.

Form:
```
Res = VDU( code, handle, &row, &col )
D0          D1    D2        D3    D4
```

Argument: code - operation code
handle - value returned from initialization call
&row - address of the variable holding the row
&col - address of the variable holding the column

Description:

The VDU routine has the following specification. Before the routine can be used, the system must be initialized. This is done by calling the VDU function with the vdu_init call; this will turn the console handler into single character mode and initialize the vdu as required. The console handler must be turned back into normal mode afterwards by using the

vdu_uninit call. The value returned from the initialization call is a handle which must be passed when making any further calls as the second argument. The first argument is always the code for the required operation, and the other two arguments are the addresses of the variables holding the physical row and column positions. These are updated suitably if the implementation of a particular feature causes the cursor to be moved from the current physical position. If the cursor is moved as a side effect, then it will not be restored; it is the responsibility of the caller to move it back if required. A common programming technique is to maintain the logical cursor position and the physical position. The values whose addresses are passed to the VDU routine represent the physical location. When the cursor is to be displayed for a long period it is placed back at the logical position.

The different calls available are listed in Chapter 7 of the *Tripos Programmer's Reference Manual*.

Quick Reference Card

File Handling:

Close	Input	Rename
CreateDir	IoErr	Seek
CurrentDir	IsInteractive	SetComment
DeleteFile	Lock	SetProtection
DupLock	Open	Unlock
Examine	Output	WaitForChar
ExNext	ParentDir	Write
Info	Read	

Task Handling:

CreateProc	Delay	Exit
DateStamp	DeviceProc	

Loading Code:

Execute	LoadSeg	UnloadSeg

Miscellaneous:

VDU

Close	To close a file for input or output.
CreateDir	To create a new directory.
CreateProc	To create a new task (process).
CurrentDir	To make a directory associated with a lock the current working directory.
DateStamp	To obtain the date and time in internal format.
Delay	To delay a task for a specified time.
DeleteFile	To delete a file or directory.
DeviceProc	To return the task (process) identifier of the task handling that I/O.
DupLock	To duplicate a lock.
Examine	To examine a directory or file associated with a lock.
Execute	To execute a CLI command.
Exit	To exit from a program.
ExNext	To examine the next entry in a directory.
Info	To return information about the disk.
Input	To identify the initial input file handle.
IoErr	To return extra information from the system.
IsInteractive	To discover whether a file is connected to a virtual terminal or not.

LoadSeg	To load a load module into memory.
Lock	To lock a file or directory.
Open	To open a file for input or output.
Output	To identify the initial output file handle.
ParentDir	To obtain the parent of a directory or file.
Read	To read bytes of data from a file.
Rename	To rename a file or directory.
Seek	To move to a logical position in a file.
SetComment	To set a comment.
SetProtection	To set file, or directory, protection.
UnloadSeg	To unload a segment previously loaded by LoadSeg.
Unlock	To unlock a file or directory.
VDU	To set up full screen support.
WaitForChar	To indicate whether characters arrive within a time limit or not.
Write	To write bytes of data to a file.

Chapter 4: The Macro Assembler

This chapter describes the Tripos Macro Assembler. It gives a brief introduction to the 68000 microchip. This chapter is intended for the reader who is acquainted with an assembly language on another computer.

Table of Contents

4.1 Introduction to the 68000 Microchip

This section gives a brief introduction to the 68000 microchip. It should help you to understand the concepts introduced later in the chapter. It assumes that you have already had experience with assembly language on another computer.

The memory available to the 68000 consists of

- the internal registers (on the chip), and
- the external main memory.

There are 17 registers, but only 16 are available at any given moment. Eight of them are **data registers** named D0 to D7, and the others are **address registers** called A0 to A7. Each register contains 32 bits. In many contexts, you may use either kind of register, but others demand a specific kind. For instance, you may use any register for operations on **word** (16 bit) and **long word** (32 bit) quantities or for indexed addressing of main memory. Although, for operations on **byte** (8 bit) operands, you may only use data registers, and for addressing main memory, you may only use address registers as stack pointers or base registers. Register A7 is the stack pointer, this is in fact two distinct registers; the system stack pointer available in supervisor mode and the user stack pointer available in user mode.

The main memory consists of a number of bytes of memory. Each byte has an identifying number called its **address**. Memory is usually (but not always) arranged so that its bytes have addresses 0, 1, 2, ..., N-2, N-1 where there are N bytes of memory in total. The size of memory that you can directly access is very large - up to 16 million bytes. The 68000 can perform operations on bytes, words, or long words of memory. A word is two consecutive bytes. In a word, the first byte has an even address. A long word is four consecutive bytes also starting at an even address. The address of a long word is the even address of its lowest numbered first byte.

As well as holding items of data being manipulated by the computer, the main memory also holds the **instructions** that tell the computer what to do. Each instruction occupies from one to 5 words, consisting of an

operation word between zero and four operand words. The operation word specifies what action is to be performed (and implicitly how many words there are in the whole instruction). The **operand words** indicate where in the registers or main memory the items to be manipulated are, and where the result should be placed.

The assembler usually executes instructions one at a time in the order that they occur in memory, like the way you follow the steps in a recipe or play the notes in a piece of written music. There is a special register called the **program counter** (PC) which you use to hold the address of the instruction you want the assembler to execute next. Some instructions, called **jumps** or **branches**, upset the usual order, and force the assembler to continue executing the instruction at a specific address. This lets the computer perform an action repeatedly, or do different things depending on the values of data items.

To remember particular things about the state of the computer, you can use one other special register called the **status register** (SR).

4.2 Calling the Assembler

The command template for assem is

PROG=FROM/A,TO/K,VER/K,LIST/K,HDR/K,EQU/K,OPT/K,INC/K

Alternatively, the format of the command line can be described as

```
assem <source file>     [TO <object file>]
                        [LIST <listing file>]
                        [VER <verification file>]
                        [HDR <header file>]
                        [EQU <equate file>]
                        [OPT <options>]
                        [INC <include dirlist>]
```

If the assembler cannot find the source filename you have specified, it appends a ".asm" suffix to the filename you supplied and tries again. By default, it produces an object file with a ".obj" suffix although it is possible to inhibit this (see the list of options at the end of this section).

Thus to assemble "fred.asm" to produce "fred.obj", you need only type

```
assem fred
```

The TO keyword can be used to override the assembler's default choice of name for the object file being produced. Thus

```
assem fred TO ofile
```

assembles fred.asm, producing binary output in "ofile" rather than in "fred.obj".

The assembler does not produce a listing file by default. It can be made to do so by specifying a file via the LIST option or by using the L option (see later) which produces a file with a ".lst" suffix. Thus

```
assem fred LIST lstfile
```

assembles "fred.asm", produces a binary file "fred.obj" and a listing file "listfile." However,

```
assem fred OPT L
```

does the same except the listing file is "fred.lst" rather than "lstfile."

As the assembler is running, it generates diagnostic messages (errors, warnings, and assembly statistics) and sends them to the screen unless you specify a verification file.

To force the inclusion of the named file in the assembly at the head of the source file, you use HDR <filename> on the command line. This has the same effect as using

```
INCLUDE "<filename>"
```

on line 1 of the source file.

To set up the list of directories that the assembler should search for any INCLUDEd files, you use the INC keyword. You should specify as many directories as you require after the INC, separating the directory names

by a comma (,), a plus sign (+), or a space. Note that if you use a space, you must enclose the entire directory list in double quotes (").

The order of the list determines the order of the directories where the assembler should search for INCLUDEd files. The assembler initially searches the current directory before any others. Thus any file that you INCLUDE in a program must be in the current directory, or in one of the directories listed in the INC list. For instance, if the program 'fred' INCLUDEs, apart from files in the current directory, a file from the directory 'intrnl/incl', a file from the directory 'include/asm', and a file from the directory 'extrnl/incl', you can give the INC directory list in these three ways:

```
assem fred INC intrnl/incl,include/asm,extrnl/incl
assem fred INC intrnl/incl+include/asm+extrnl/incl
assem fred INC "intrnl/incl include/asm extrnl/incl"
```

The EQU keyword allows equated symbols to be extracted from the source to the specified file. This can be used to help build a header file from a program sprinkled with equates.

The OPT keyword allows you to pass certain options to the assembler. Each option consists of a single character (in either upper or lower case), possibly followed immediately by a number. Valid options follow here:

- S produces a symbol dump as a part of the object file.
- D inhibits the dumping of local labels as part of a symbol dump. (For C programmers, any label beginning with a period is considered as a local label).
- C ignores the distinction between upper and lower case in labels.
- X produces a cross-reference table at the end of the listing file.
- L produces a listing file with the default suffix (".lst").
- N inhibits production of object files.

Examples

```
assem fred.asm TO fred.o
```

assembles the file fred.asm and produces an object module in the file fred.o.

```
assem fred OPT LSX
```

assembles the file fred.asm, produces an object module in the file fred.obj, which includes a symbol dump, and produces a listing file in fred.lst, which also contains a cross-reference listing.

4.3 Program Encoding

A program acceptable to the assembler takes the form of a series of input lines that can include any of the following:

- Comment or Blank lines
- Executable Instructions
- Assembler Directives

4.3.1 Comments

To introduce comments into the program, you can use three different methods:

1. Type a semicolon (;) anywhere on a line and follow it with the text of the comment. For example,

   ```
   CMPA.L A1,A2  ; Are the pointers equal?
   ```

2. Type an asterisk (*) in column one of a line and follow it with the text of the comment. For example,

   ```
   * This entire line is a comment
   ```

3. Follow any complete instruction or directive with a least one space and some text. For example,

```
MOVEQ #IO,D0 place initial value in D0
```

In addition, note that all blank lines are treated by the assembler as comment lines.

4.3.2 Executable Instructions

The source statements have the general overall format:

[<label>] <opcode> [<operand>[,<operand>]...][<comment>]

To separate each field from the next, press the SPACEBAR or TAB key. This produces a separator character. You may use more than one space to separate fields.

4.3.2.1 Label Field

A label is a user symbol, or programmer-defined name, that either

a) Starts in the first column and is separated from the next field by at least one space,

or

b) Starts in any column, and is followed immediately with a colon (:).

If a label is present, then it must be the first non-blank item on the line. The assembler assigns the value and type of the program counter, that is, the memory address of the first byte of the instruction or data being referenced, to the label. Labels are allowed on all instructions, and on some directives, or they may stand alone on a line. See the specifications of individual directives in Section 4.7 for whether a label field is allowed.

Note: You must not give multiple definitions to labels. Also, you must not use instruction names, directives, or register names as labels.

4.3.2.2 Local Labels

Local labels are provided as an extension to the MOTOROLA specification. They take the form nnn$ and are only valid between any proper (named) labels. Thus, in this example code segment

```
Labels      Opcodes     Operands

FOO:        MOVE.L      D6,D0
1$:         MOVE.B      (A0)+,(A1)+
            DBRA        D0,1$
            MOVEQ       #20,D0
BAA:        TRAP        #4
```

the label 1$ is only available from the line following the one labelled FOO to the line before the one labelled BAA. In this case, you could then use the label 1$ in a different scope elsewhere in the program.

4.3.2.3 Opcode Field

The Opcode field follows the Label field and is separated from it by at least one space. Entries in this field are of three types.

1. The MC68000 operation codes, as defined in the *MC68000 User Manual*.

2. Assembler Directives.

3. Macro invocations.

To enter instructions and directives that can operate on more than one data size, you use an optional Size-Specifier subfield, which is separated from the opcode by the period (.) character. Possible size specifiers are as follows:

B - Byte-sized data (8 bits)
W - Word-sized data (16 bits)
L - Long Word-sized data (32 bits)
or Long Branch specifier
S - Short Branch specifier

The size specifier must match with the instruction or directive type that you use.

4.3.2.4 Operand Field

If present, the operand field contains one or more operands to the instruction or directive, and must be separated from it by at least one space. When you have two or more operands in the field, you must separate them with a comma (,). The operand field terminates with a space or newline character (a newline character is what the assembler receives when you press RETURN), so you must not use spaces between operands.

4.3.2.5 Comment Field

Anything after the terminating space of the operand field is ignored. So the assembler treats any characters you insert after a space as a comment.

4.4 Expressions

An expression is a combination of symbols, constants, algebraic operators, and parentheses that you can use to specify the operand field to instructions or directives. You may include relative symbols in expressions, but they can only be operated on by a subset of the operators.

4.4.1 Operators

The available operators are listed below in order of precedence.

1. Monadic Minus, Logical NOT (- and ~)
2. Lshift, Rshift (< < and > >)
3. Logical AND, Logical OR (& and !)
4. Multiply, Divide (* and /)
5. Add, Subtract (+ and -)

To override the precedence of the operators, enclose sub-expressions in parentheses. The assembler evaluates operators of equal precedence from left to right. Note that you should not have any spaces in an expression: a space is regarded as a delimiter between fields.

4.4.2 Operand Types for Operators

In the following table, 'A' represents absolute symbols, and R represents relative symbols. The table shows all the possible operator/operand combinations, with the type of the resulting value. 'x' indicates an error. The Monadic minus and the Logical not operators are only valid with an absolute operand.

Operators	Operands			
	A op A	R op R	A op R	R op A
+	A	x	R	R
–	A	A	x	R
*	A	x	x	x
/	A	x	x	x
&	A	x	x	x
!	A	x	x	x
>>	A	x	x	x
<<	A	x	x	x

Table 4-A: Operand Types for Operators

4.4.3 Symbols

A **symbol** is a string of up to 30 characters. The first character of a symbol must be one of following:

- An alphabetic charcter (a-z, or A-Z).
- An underscore (_).
- A period (.).

The rest of the characters in the string can be any of these characters or also numeric (0 through 9). In all symbols, the lower case characters (a-z) are not treated as synonyms with their upper case equivalents (unless you use the option C when you invoke the assembler). So 'fred' is different from 'FRED' and 'FRed'. However, the assembler recognizes instruction opcodes, directives, and register names in either upper or lower case. A label equated to a register name with EQUR is also recognized by the assembler in either upper or lower case. Symbols can be up to 30 characters in length, all of which are significant. The assembler takes symbols longer than this and truncates them to 30 characters, giving a warning that it has done so. The Instruction names, Directive names, Register names, and special symbols CCR, SR, SP and USP cannot be used as user symbols. A symbol can be one of three types:

Absolute

a) The symbol was SET or EQUated to an Absolute value

Relative

a) The symbol was SET or EQUated to a Relative value

b) The symbol was used as a label

Register

a) The symbol was set to a register name using EQUR (This is an extension from the MOTOROLA specification)

There is a special symbol *, which has the value and type of the current program counter, that is, the address of the current instruction or directive that the assembler is acting on.

4.4.4 Numbers

You may use a number as a term of an expression, or as a single value. Numbers ALWAYS have absolute values and can take one of the following formats:

Decimal
(a string of decimal digits)

Example: 1234

Hexadecimal
($ followed by a string of hex digits)

Example: $89AB

Octal
(@ followed by a string of octal digits)

Example: @743

Binary
(% followed by zeros and ones)

Example: %10110111

ASCII Literal
(Up to 4 ASCII characters within quotes)

Examples: 'ABCD' '*'

Strings of less than 4 characters are justified to the right, using NUL as the packing character.

To obtain a quote character in the string, you must use two quotes. An example of this is

```
'It''s'
```

4.5 Addressing Modes

The effective address modes define the operands to instructions and directives, and you can find a detailed description of them in any good reference book on the 68000. Addresses refer to individual bytes, but instructions, Word and Long Word references, access more than one byte, and the address for these must be word aligned.

In the following table, Dn represents one of the data registers (D0-D7), 'An' represents one of the address registers (A0-A7, SP and PC), 'a' represents an absolute expression, 'r' represents a relative expression, and 'Xn' represents An or Dn, with an optional '.W' or '.L' size specifier. The syntax for each of the modes is as follows:

Table 4-B: Macro Assembler Address Modes and Registers

Address Mode	Description and Examples
Dn	Data Register Direct Example: MOVE D0,D1
An	Address Register Direct Example: MOVEA A0,A1
(An)	Address Register Indirect Example: MOVE D0,(A1)
(An)+	Address Register Indirect Post Increment Example: MOVE (A7)+,D0
-(An)	Address Register Indirect Pre Decrement Example: MOVE D0,-(A7)
a(An)	Address Register Indirect with (16-bit) Displacement Example: MOVE 20(A0),D1

a(An,Xn)	Address Register Indirect with Index
	(a is an 8-bit Displacement)
	Example: MOVE 0(A0,D0),D1
	MOVE 12(A1,A0.L),D2
	MOVE 120(A0,D6.W),D4

(continuation of 4-B)

Address Mode	Description and Examples
a	Short absolute (16 bits) Example: MOVE $1000,D0
a	Long absolute (32 bits) Example: MOVE $10000,D0
r	Program Counter Relative with Displacement (when label is already defined) Example: MOVE ABC,D0 (ABC is relative and already defined) Note that when an instruction such as MOVE ABC,D0 is encountered the assembler will use the "program counter relative with displacement" mode whenever the symbol used has already been defined.
r	Long absolute with Relocation Example: MOVE ABC,D0 (ABC is relative but yet to be defined)
r(PC)	Program Counter Relative with (16-bit) Displacement Example: MOVE ABC(PC),D1 (ABC is relative and already defined) Example: MOVE DEF(PC),D0 (DEF is relative but yet to be defined)

(continuation of 4-B)

Address Mode	Description and Examples
r(Xn)	Program Counter Relative with Index (where r is an 8-bit relocatable symbol. Note that this is shorthand form of r(PC,Xn)) Example: MOVE ABC(D0.L),D1 (ABC is relative)
r(PC,Xn)	Program Counter Relative with Index (where r is an 8-bit relocatable symbol) Example: MOVE ABC(PC,D0.L),D1 (ABC is relative)
#a	Immediate data Example: MOVE #1234,D0

USP)
CCR)
SR)
	Special addressing modes
	Example: MOVE A0,USP
	MOVE D0,CCR
	MOVE D1,SR

4.6 Variants on Instruction Types

Certain instructions (for example, ADD, CMP) have an **address variant** (that refers to address registers as destinations), **immediate** and **quick** forms (when immediate data possibly within a restricted size range appears as an operand), and a **memory variant** (where both operands must be a postincrement address).

To force a particular variant to be used, you may append A, Q, I or M to the instruction mnemonic. In this case, the assembler uses the specified form of the instruction, if it exists, or gives an error message.

If, however, you specify no particular variant, the assembler automatically converts to the 'I', 'A' or 'M' forms where appropriate. However, it does not convert to the 'Q' form. For example, the assembler converts the following:

```
      ADD.L    A2,A1
```

to

```
      ADDA.L   A2,A1
```

4.7 Directives

All assembler directives (with the exception of DC and DCB) are instructions to the assembler, rather than instructions to be translated into object code. At the beginning of this section, there is a list of all the directives (Table 4-C), arranged by function; at the end there is an individual decription for each directive, arranged by function.

Note that the assembler only allows labels on directives where specified. For example, EQU is allowed a label. It is optional for RORG, but not allowed for LLEN or TTL.

The following table lists the directives by function:

Table 4-C: Directives

Assembly Control

Directive	Description
SECTION	Program section
RORG	Relocatable origin
OFFSET	Define offsets
END	Program end

Symbol Definition

Directive	Description
EQU	Assign permanent value
EQUR	Assign permanent register value
REG	Assign permanent value
SET	Assign temporary value

Data Definition

Directive	Description
DC	Define constants
DCB	Define Constant Block
DS	Define storage

(continuation of 4-C)

Listing Control

Directive	Description
PAGE	Page-throw to listing
LIST	Turn on listing
NOLIST (NOL)	Turn off listing
SPC n	Skip n blank lines
NOPAGE	Turn off paging
LLEN n	Set line length (60 < = n < = 132)
PLEN n	Set page length (24 < = n < = 100)
TTL	Set program title (max 80 chars)
NOOBJ	Disable object code output
FAIL	Generate an assembly error
FORMAT	No action
NOFORMAT	No action

Conditional Assembly

Directive	Description
CNOP	Conditional NOP for alignment
IFEQ	Assemble if expression is 0
IFNE	Assemble if expression is not 0
IFGT	Assemble if expression > 0
IFGE	Assemble if expression > = 0
IFLT	Assemble if expression < 0
IFLE	Assemble if expression < = 0
IFC	Assemble if strings are identical
IFNC	Assemble if strings are not identical
IFD	Assemble if symbol is defined
IFND	Assemble if symbols is not defined
ENDC	End of conditional assembly

Macro Directives

Directive	Description
MACRO	Define a macro name
NARG	Special symbol
ENDM	End of macro definition
MEXIT	Exit the macro expansion

External Symbols

Directive	Description
XDEF	Define external name
XREF	Reference external name

General Directives

Directive	Description
INCLUDE	Insert file in the source
MASK2	No action
IDNT	Name program unit

Assembly Control Directives

SECTION Program Section

Format: [<label>] SECTION <name>[,<type>]

This directive tells the assembler to restore the counter to the last location allocated in the named section (or to zero if used for the first time).

<name> is a character string optionally enclosed in double quotes.
<type> if included, must be one of the following keywords:

CODE	indicates that the section contains relocatable code. This is the default.
DATA	indicates that the section contains initialised data (only).
BSS	indicates that the section contains uninitialised data

The assembler can maintain up to 255 sections. Initially, the assembler begins with an unnamed CODE section. The assembler assigns the optional symbol <labels> to the value of the program counter after it has executed the SECTION directive. In addition, where a section is unnamed, the shorthand for that section is the keyword CODE.

RORG Set Relative Origin

Format: [<label>] RORG <absexp>

The RORG directive changes the program counter to be <absexp> bytes from the start of the current relocatable section. The assembler assigns relocatable memory locations to subsequent statements, starting with the value assigned to the program counter. To do addressing in relocatable sections, you use the 'program counter relative with displacement' addressing mode. The label value assignment is the same as for SECTION.

OFFSET Define offsets

Format: OFFSET <absexp>

To define a table of offsets via the DS directive beginning at the address <absexp>, you use the OFFSET directive. Symbols defined in an OFFSET table are kept internally, but no code-producing intructions or directives may appear. To terminate an OFFSET section, you use a RORG, OFFSET, SECTION, or END directive.

END End of program

Format: [<label>] END

The END directive tells the assembler that the source is finished, and the assembler ignores subsequent source statements. When the assembler encounters the END directive during the first pass, it begins the second pass. If, however, it detects an end-of-file before an END directive, it gives a warning message. If the label field is present, then the assembler assigns the value of the current program counter to the label before it executes the END directive.

Symbol Definition Directives

EQU Equate symbol value

Format: <label> EQU <exp>

The EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The value assigned is permanent, so you may not define the label anywhere else in the program.

Note: Do not insert forward references within the expression.

EQUR Equate register value

Format: <label> EQUR <register>

This directive lets you equate one of the processor registers with a user symbol. Only the Address and Data registers are valid, so special symbols like SR, CCR, and USP are illegal here. The register is permanent, so you cannot define the label anywhere else in the program. The register must not be a forward reference to another EQUR statement. The assembler matches labels defined in this way without distinguishing upper and lower case.

REG Define register list

Format: <label> REG <register list>

The REG directive assigns a value to label that the assembler can translate into the register list mask format used in the MOVEM instruction. <register list> is of the form

R1 [-R2] [/R3 [-R4]]...

SET Set symbol value

Format: <label> SET <exp>

The SET directive assigns the value of the expression in the operand field to the symbol in the label field. SET is identical to EQU, apart from the fact that the assignment is temporary. You can always change SET later on in the program.

Note: You should not insert forward references within the expression or refer forward to symbols that you defined with SET.

Data Definition Directives

DC Define Constant

Format: [<label>] DC[.<size>] <list>

The DC directive defines a constant value in memory. It may have any number of operands, separated by commas (,). The values in the list must be capable of being held in the data location whose size is given by the size specifier on the directive. If you do not give a size specifier, DC assumes it is .W. If the size is .B, then there is one other data type that can be used: that of the ASCII string. This is an arbitrarily long series of ASCII characters, contained within quotation marks. As with ASCII literals, if you require a quotation mark in the string, then you must enter two. If the size is .W or .L, then the assembler aligns the data onto a word boundary.

DCB Define Constant Block

Format: [<label>] DCB[.<size>] <absexp>,<exp>

You use the DCB directive to set a number (given by <absexp>) of bytes, words, or longwords to the value of the expression <exp>. DCB.<size> n,exp is equivalent to repeating n times the statement DC.<size> exp.

DS Define Storage

Format: [<label>] DS[.<size>] <absexp>

To reserve memory locations, you use the DS directive. DS, however, does no initialisation. The amount of space the assembler allocates depends on the data size (that you give with the size specifier on the directive), and the value of the expression in the operand field. The assembler interprets this as the number of data items of that size to allocate. As with DC, if the size specifier is .W or .L, DS aligns the space onto a word boundary. So, DS.W 0 has the effect of aligning to a word boundary only. If you do not give a size specifier, DS assumes a default of .W. See CNOP for a more general way of handling alignment.

Listing Control Directives

PAGE Page Throw

Format: PAGE

Unless paging has been inhibited, PAGE advances the assembly listing to the top of the next page. The PAGE directive does not appear on the output listing.

LIST Turn on Listing

Format: LIST

The LIST directive tells the assembler to produce the assembly listing file. Listing continues until it encounters either an END or a NOLIST directive. This directive is only active when the assembler is producing a listing file. The LIST directive does not appear on the output listing.

NOLIST Turn off Listing

Format: NOLIST
NOL

The NOLIST or NOL directive turns off the production of the assembly listing file. Listing ceases until the assembler encounters either an END or a LIST directive. The NOLIST directive does not appear on the program listing.

SPC Space Blank Lines

Format: SPC <number>

The SPC directive outputs the number of blank lines given by the operand field, to the assembly listing. The SPC directive does not appear on the program listing.

NOPAGE Turn off Paging

Format: NOPAGE

The NOPAGE directive turns off the printing of page throws and title headers on the assembly listing.

LLEN Set Line Length

Format: LLEN <number>

The LLEN directive sets the line length of the assembly listing file to the value you specified in the operand field. The value must lie between 60 and 132, and can only be set once in the program. The LLEN directive does not appear on the assembly listing. The default is 132 characters.

PLEN Set Page Length

Format: PLEN <number>

The PLEN directive sets the page length of the assembly listing file to the value you specified in the operand field. The value must lie between 24 and 100, and you can only set it once in the program. The default is 60 lines.

TTL Set Program Title

Format: TTL <title string>

The TTL directive sets the title of the program to the string you gave in the operand field. This string appears as the page heading in the assembly listing. The string starts at the first non-blank character after the TTL, and continues until the end of line. It must not be longer than 40 characters in length. The TTL directive does not appear on the program listing.

NOOBJ Disable Object Code Generation

Format: NOOBJ

The NOOBJ directive disables the production of the object code file at the end of assembly. This directive disables the production of the code file, even if you specified a file name when you called the assembler.

FAIL Generate a user error

Format: FAIL

The FAIL directive tells the assembler to flag an error for this input line.

FORMAT No action

Format: FORMAT

The assembler accepts this directive but takes no action on receiving it. FORMAT is included for compatibility with other assemblers.

NOFORMAT No action

Format: NOFORMAT

The assembler accepts this directive but takes no action on receiving it. NOFORMAT is included for compatibility with other assemblers.

Conditional Assembly Directives

CNOP Conditional NOP

Format: [<label>] CNOP <number>,<number>

This directive is an extension from the Motorola standard and allows a section of code to be aligned on any boundary. In particular, it allows any data structure or entry point to be aligned to a long word boundary.

The first expression represents an offset, while the second expression represents the alignment required for the base. The code is aligned to the specified offset from the nearest required alignment boundary. Thus

```
CNOP    0,4
```

aligns code to the next long word boundary while

```
CNOP    2,4
```

aligns code to the word boundary 2 bytes beyond the nearest long word aligned boundary.

IFEQ	Assemble if expresion = 0
IFNE	Assemble if expression < > 0
IFGT	Assemble if expression > 0
IFGE	Assemble if expression > = 0
IFLT	Assemble if expression < 0
IFLE	Assemble if expression < = 0

Format: IFxx <absexp>

You use the IFxx range of directives to enable or disable assembly, depending on the value of the expression in the operand field. If the condition is not TRUE (for example, IFEQ 2+1), assembly ceases (that is, it is disabled). The conditional assembly switch remains active until the assembler finds a matching ENDC statement. You can nest conditional assembly switches arbitrarily, terminating each level of nesting with a matching ENDC.

IFC	Assemble if strings are identical
IFNC	Assemble if strings are not identical

Format: IFC <string>,<string>
IFNC <string>,<string>

The strings must be a series of ASCII characters enclosed in single quotes, for example, 'FOO' or '' (the empty string). If the condition is not TRUE, assembly ceases (that is, it is disabled). Again the conditional assembly switch remains active until the assembler finds a matching ENDC statement.

IFD Assemble if symbol defined
IFND Assemble if symbol not defined

Format: IFD <symbol name>
IFND <symbol name>

Depending on whether or not you have already defined the symbol, the assembler enables or disables assembly until it finds a matching ENDC.

ENDC End conditional assembly

Format: ENDC

To terminate a conditional assembly, you use the ENDC directive, set up with any of the 8 IFxx directives above. ENDC matches the most recently encountered condition directive.

Macro Directives

MACRO Start a macro definition

Format: <label> MACRO

MACRO introduces a macro definition. ENDM terminates a macro definition. You must provide a label, which the assembler uses as the name of the macro; subsequent uses of that label as an operand expand the contents of the macro and insert them into the source code. A macro can contain any opcode, most assembler directives, or any previously defined macro. A plus (+) sign in the listing, marks any code generated by macro expansion. When you use a macro name, you may append a number of arguments, separated by commas. If the argument contains a space (for example, a string containing a space) then you must enclose the entire argument within < (less than) and > (greater than) symbols.

The assembler stores up and saves the source code that you enter (after a MACRO directive and before an ENDM directive) as the contents of the macro. The code can contain any normal source code. In addition, the symbol \ (backslash) has a special meaning. Backslash followed by a number n indicates that the value of the nth argument is to inserted into the code. If the nth argument is omitted then nothing is inserted. Backslash followed by the symbol '@' tells the assembler to generate the text '.nnn', where nnn is the number of times the \@ combination has been encountered. This is normally used to generate unique labels within a macro.

You may not nest macro definitions, that is, you cannot define a macro within a macro, although you can call a macro you previously defined. There is a limit to the level of nesting of macro calls. This limit is currently set at ten.

Macro expansion stops when the assembler encounters the end of the stored macro text, or when it finds a MEXIT directive.

NARG Special symbol

Format: NARG

The symbol NARG is a special reserved symbol and the assembler assigns it the index of the last argument passed to the macro in the parameter list (even nulls). Outside of a macro expansion, NARG has the value 0.

ENDM Terminate a macro definition

Format: ENDM

This terminates a macro definition introduced by a MACRO directive.

MEXIT Exit from macro expansion

Format: MEXIT

You use this directive to exit from macro expansion mode, usually in conjunction with the IFEQ and IFNE directives. It allows conditional expansion of macros. Once it has executed the directive, the assembler stops expanding the current macro as though there were no more stored text to include.

External Symbols

XDEF Define an internal label as an external entry point

Format: XDEF <label> [,<label>...]

One or more absolute or relocatable labels may follow the XDEF directive. Each label defined here generates an external symbol definition. You can make references to the symbol in other modules (possibly from a high-level language) and satisfy the references with a linker. If you use this directive or XREF, then you cannot directly execute the code produced by the assembler.

XREF Define an external name

Format: XREF <label> [,<label>...]

One or more labels that must not have been defined elsewhere in the program follow the XREF directive. Subsequent uses of the label tell the assembler to generate an external reference for that label. You use the label as if it referred to an absolute or relocatable value depending on whether the matching XDEF referred to an absolute or relocatable symbol.

The actual value used is filled in from another module by the linker. The linker also generates any relocation information that may be required in order for the resulting code to be relocatable.

External symbols are normally used as follows. To specify a routine in one program segment as an external definition, you place a label at the start of the routine and quote the label after an XDEF directive. Another program may call that routine if it declares a label via the XREF directive and then jumps to the label so declared.

INCLUDE Insert an external file

Format: INCLUDE "<file name>"

The INCLUDE directive allows the inclusion of external files into the program source. You set up the file that INCLUDE inserts with the string descriptor in the operand field. You can nest INCLUDE directives up to a depth of three, enclosing the file names in quotes as shown. INCLUDE is especially useful when you require a standard set of macro definitions or EQUs in several programs.

You can place the definitions in a single file and then refer to them from other programs with a suitable INCLUDE. It is often convenient to place NOLIST and LIST directives at the head and tail of files you intend to include via INCLUDE. Tripos searches for the file specification first in the current directory, then in each subsequent directory in the list you gave in the INC option.

MASK2 No action

Format: MASK2

The assembler accepts the MASK2 directive, but it takes no action on receiving it.

IDNT Name program unit

Format: IDNT <string>

A program unit, which consists of one or more sections, must have a name. Using the IDNT directive, you can define a name consisting of a string optionally enclosed in double quotes. If the assembler does not find a IDNT directive, it outputs a program unit name that is a null string.

Chapter 5: The Linker

This chapter describes the Tripos Linker. The Tripos Linker produces a single binary load file from one or more input files. It can also produce overlaid programs.

Table of Contents

5.1 Introduction

ALINK produces a single binary output file from one or more input files. These input files, known as **object files**, may contain external symbol information. To produce object files, you use your assembler or language translator. Before producing the output, or **load file**, the linker resolves all references to symbols.

The linker can also produce a link map and symbol cross reference table.

Associated with the linker is an **overlay supervisor**. You can use the overlay supervisor to overlay programs written in a variety of languages. The linker produces load files suitable for overlaying in this way.

You can drive the linker in two ways:

1. from a **Command line**. You can specify most of the information necessary for running the linker in the command parameters.

2. from a **Parameter file**. As an alternative, if a program is being linked repetitively, you can use a parameter file to specify all the data for the linker.

These two methods can take three types of input files:

1. **Primary binary input**. This refers to one or more object files that form the initial binary input to the linker. These files are always output to the load file, and the primary input must not be empty.

2. **Overlay files**. If overlaying, the primary input forms the root of the overlay tree, and the overlay files form the rest of the structure.

3. **Libraries**. This refers to specified code that the linker incorporates automatically. Libraries may be resident or scanned. A **resident library** is a load file which may be resident in memory, or loaded as part of the 'library open' call in the operating system. A **scanned library** is an object file within an archive format file. The linker only loads the file if there are any outstanding external references to the library.

The linker works in two passes.

1. In the first pass, the linker reads all the primary, library and overlay files, and records the code segments and external symbol information. At the end of the first pass, the linker outputs the map and cross reference table, if required.

2. If you specify an output file, then the linker makes the second pass through the input. First it copies the primary input files to the output, resolving symbol references in the process, and then it copies out the required library code segments in the same way. Note that the library code segments form part of the root of the overlay tree. Next, the linker produces data for the overlay supervisor, and finally outputs the overlay files.

In the first pass, after reading the primary and overlay input files, the linker inspects its table of symbols, and if there are any remaining unresolved references, it reads the files, if any, that you specified as the library input. The linker then marks any code segments containing external definitions for these unresolved references for subsequent inclusion in the load file. The linker only includes those library code segments that you have referenced.

5.2 Using the Linker

To use the linker, you must know the command syntax, the type of input and output that the linker uses, and the possible errors that may occur. These are explained here.

5.2.1 Command Line Syntax.

The ALINK command has the following parameters:

```
ALINK [FROM | ROOT] files [TO file] [WITH file]
  [VER file] [LIBRARY | LIB files] [MAP file]
  [XREF file] [WIDTH n] [SMALL]
```

where 'file' means a single file name, 'files' means zero or more file names, separated by a comma or plus sign, and 'n' is an integer.

The keyword template is

```
"FROM=ROOT,TO/K,WITH/K,VER/K,LIBRARY=LIB/K,
 MAP/K,XREF/K,WIDTH/K,SMALL/S"
```

The following are examples of valid uses of the ALINK command:

```
ALINK a
ALINK ROOT a+b+c+d MAP map-file WIDTH 120
ALINK a,b,c TO output LIBRARY :flib/lib,obj/newlib
```

When you give a list of files, the linker reads them in the order you specify.

The parameters have the following meanings:

FROM: specifies the object files that you want as the primary binary input. The linker always copies the contents of these files to the load file to form part of the overlay root. At least one primary binary input file must be specified. ROOT is a synonym for FROM.

TO: specifies the destination for the load file. If this parameter is not given, the linker omits the second pass.

WITH: specifies files containing the linker parameters, for example, normal command lines. Usually you only use one file here, but, for completeness, you can give a list of files. Note that parameters on the command line override those in WITH files. You can find a full description of the syntax of these files in section 5.2.2 of this manual.

VER: specifies the destination of messages from the linker. If you do not specify VER, the linker sends all messages to the standard output (usually the terminal).

LIBRARY: specifies the files that you want to be scanned as the library. The linker includes only referenced code segments. LIB is a valid alternative for LIBRARY.

MAP: specifies the destination of the link map.

XREF: specifies the destination of the cross reference output.

WIDTH: specifies the output width that the linker can use when producing the link map and cross reference table. For example, if you send output to a printer, you may need this parameter.

SMALL optimizes the use of space during linking. If you use this switch, then you must sacrifice speed; SMALL may use less space than usual, but it is slow. You would use SMALL if your previous attempt to link failed through lack of memory.

5.2.2 WITH Files

WITH files contain parameters for the linker. You use them to save typing a long and complex ALINK command line many times.

A WITH file consists of a series of parameters, one per line, each consisting of a keyword followed by data. You can terminate lines with a semicolon (;), where the linker ignores the rest of the line. You can then use the rest of the line after the semicolon to include a comment. The linker ignores blank lines.

The keywords available are as follows:

```
FROM (or ROOT)  files
TO              file
LIBRARY         files
MAP             [file]
XREF            [file]
OVERLAY
tree specification
#
WIDTH           n
```

where 'file' is a single filename, 'files' is one or more filenames, '[file]' is an optional filename, and 'n' is an integer. You may use an asterisk symbol (*) to split long lines; placing one at the end of a line tells the printer to read the next line as a continuation line. If the filename after MAP or XREF is omitted, the output goes to the VER file (the terminal by default).

Parameters on the command line override those in a WITH file, so that you can make small variations on standard links by combining command line parameters and WITH files. Similarly, if you specify a parameter more than once in WITH files, the linker uses the first occurrence.

Note: In the second example below, this is true even if the first value given to a parameter is null.

Examples of WITH files and the corresponding ALINK calls:

```
ALINK WITH link-file
```

where 'link-file' contains

```
FROM    obj/main,obj/s
TO      bin/test
LIBRARY obj/lib
MAP
XREF    xo
```

is the same as specifying

```
ALINK FROM obj/main,obj/s TO bin/test
      LIBRARY obj/lib XREF xo
```

The command

```
ALINK WITH lkin LIBRARY ""
```

where 'lkin' contains

```
FROM    bin/prog,bin/subs
LIBRARY nag/fortlib
TO      linklib/prog
```

is the same as the command line

```
ALINK FROM bin/prog,bin/subs TO linklib.prog
```

Note: In the example above, the null parameter for LIBRARY on the command line overrides the value 'nag/fortlib' in the WITH file, and so the linker does not read any libraries.

5.2.3 Errors and Other Exceptions

Various errors can occur while the linker is running. Most of the messages are self-explanatory and refer to the failure to open files, or to errors in command or binary file format. After an error, the linker terminates at once.

There are a few messages that are warnings only. The most important ones refer to undefined or multiply-defined symbols. The linker should not terminate after receiving a warning.

If any undefined symbols remain at the end of the first pass, the linker produces a warning, and outputs a table of such symbols. During the second pass, references to these symbols become references to **location zero**.

If the linker finds more than one definition of a symbol during the first pass, it puts out a warning, and ignores the later definition. The linker does not produce this message if the second definition occurs in a library file, so that you can replace library routines without it producing spurious messages. A serious error follows if the linker finds inconsistent symbol references, and linking then terminates at once.

Since the linker only uses the first definition of any symbol, it is important that you understand the following order in which files are read.

1. Primary (FROM or ROOT) input.
2. Overlay files.
3. LIBRARY files.

Within each group, the linker reads the files in the order that you specify in the file list. Thus definitions in the primary input override those in the overlay files, and those in the libraries have lowest priority.

5.2.4 MAP and XREF Output

The link map, which the linker produces after the first pass, lists all the code segments that the linker output to the load file in the second pass, in the order that they must be written.

For each code segment, the linker outputs a header, starting with the name of the file (truncated to eight letters), the code segment reference number, the type (that is, data, code, bss, or COMMON), and size. If the code segment was in an overlay file, the linker also gives the overlay level and overlay ordinate.

After the header, the linker prints each symbol defined in the code segment, together with its value. It prints the symbols in ascending order of their values, appending an asterisk (*) to absolute values.

The value of the WIDTH parameter determines the number of symbols printed per line. If this is too small, then the linker prints one symbol on each line.

The cross reference output also lists each code segment, with the same header as in the map.

The header is followed by a list of the symbols with their references. Each reference consists of a pair of integers, giving the offset of the reference and the number of the code segment in which it occurs. The code segment number refers to the number given in each header.

5.3 Overlaying

The automatic overlay system provided by the linker and the overlay supervisor allows programs to occupy less memory when running, without any alterations to the program structure.

When using overlaying, you should consider the program as a **tree** structure. That is, with the **root** of the tree as the primary binary input, together with library code segments and COMMON blocks. This root is always resident in memory. The overlay files then form the other nodes of the tree, according to specifications in the OVERLAY directive.

The output from the linker when overlaying, as in the usual case, is a single binary file, which consists of all the code segments, together with information giving the location within the file of each node of the overlay tree. When you load the program only the root is brought into memory. An overlay supervisor takes care of loading and unloading the overlay segments automatically. The linker includes this overlay supervisor in the output file produced from an link using overlays. The overlay supervisor is invisible to the program running.

5.3.1 OVERLAY Directive

To specify the tree structure of a program to the linker, you use the OVERLAY directive. This directive is exceptional in that you can only use it in WITH files. As with other parameters, the linker uses the first OVERLAY directive you give it.

The format of the directive is

```
OVERLAY
Xfiles
.
.
.
#
```

Note: The overlay directive can span many lines. The linker recognizes a hash (sharp sign '#') or the end-of-file as a terminator for the directive.

Each line after OVERLAY specifies one node of the tree, and consists of a count X and a file list.

The level of a node specifies its 'depth' in the tree, starting at zero, which is the level of the root. The count, X, given in the directive, consists of zero or more asterisks, and the overlay level of the node is given by X + 1.

As well as the level, each node other than the root has an **ordinate** value. This refers to the order in which the linker should read the descendents of each node, and starts at 1, for the first 'offspring' of a parent node.

Note: There may be nodes with the same level and ordinate, but with different parents.

While reading the OVERLAY directive, the linker remembers the current level, and, for each new node, compares the level specified with this value. If less, then the new node is a descendent of a previous one. If equal, the new node has the same parent as the current one. If greater, the new node is a direct descendant of the current one, and so the new level must be one greater than the current value.

A number of examples may help to clarify this:

Directive	Level	Ordinate	Tree
OVERLAY			ROOT
a	1	1	/|\
b	1	2	a b c
c	1	3	
#			
OVERLAY			ROOT
a	1	1	/\
b	1	2	a b
*c	2	1	/|
*d	2	2	c d
#			

```
OVERLAY
a               1       1
b               1       2
*c              2       1
*d              2       2
e               1       3
f               1       4
*g              2       1
*h              2       2
**i             3       1
**j             3       2
*k              2       3
l               1       5
#
```

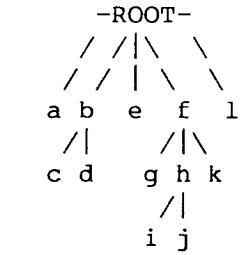


Figure 5-A

The level and ordinate values given above refer to the node specified on the same line. Note that all the files given in the examples above could have been file lists. Single letters are for clarity. For example, Figure 5-B

```
ROOT        bin/mainaaa
OVERLAY
bin/mainbbb,bin/mainccc,bin/mainddd
*bin/makereal
*bin/trbblock,bin/transint,bin/transr*
 bin/transri
bin/outcode
#
```

Figure 5-B

specifies the tree in the following figure:

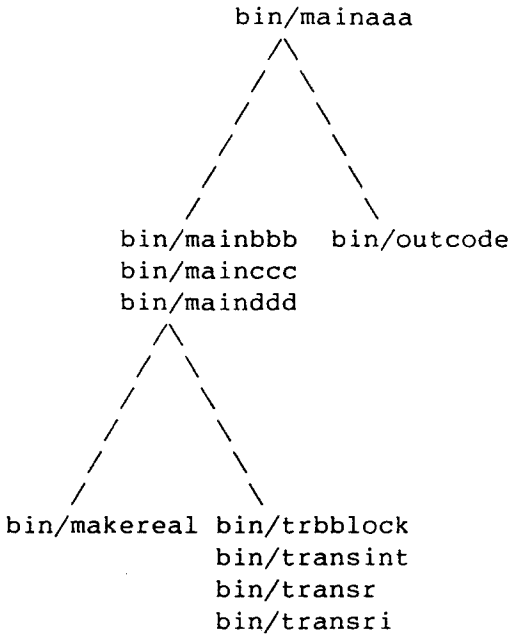


Figure 5-C

During linking, the linker reads the overlay files in the order you specified in the directive, line by line. The linker preserves this order in the map and cross reference output, and so you can deduce the exact tree structure from the overlay level and ordinate the linker prints with each code segment.

5.3.2 References To Symbols

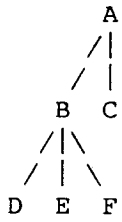
While linking an overlaid program, the linker checks each symbol reference for validity.

Suppose that the reference is in a tree node R, and the symbol in a node S. Then the reference is legal if one of the following is true:

a. R and S are the same node,
b. R is a descendent of S, **or**
c. R is the parent of S.

References of the third type above are known as **overlay references**. In this case, the linker enters the overlay supervisor when the program is run. The overlay supervisor then checks to see if the code segment containing the symbol is already in memory. If not, first the code segment, if any, at this level, and all its descendents are unloaded, and then the node containing the symbol is brought into memory. An overlaid code segment returns directly to its caller, and so is not unloaded from memory until another node is loaded on top of it.

For example, suppose that the tree is:



When the linker first loads the program, only A is in memory. When the linker finds a reference in A to a symbol in B, it loads and enters B. If B in turn calls D then again a new node is loaded. When B returns to A, both B and D are left in memory, and the linker does not reload them if the program requires them later. Now suppose that A calls C. First the linker unloads the code segments that it does not require, and which it may overwrite. In this case, these are B and D. Once it has reclaimed the memory for these, the linker can load C.

Thus, when the linker executes a given node, all the node's 'ancestors', up to the root are in memory, and possibly some of its descendents.

5.3.3 Cautionary Points

The linker assumes that all overlay references are jumps or subroutine calls, and routes them through the overlay supervisor. Thus, you should not use overlay symbols as data labels.

Try to avoid impure code when overlaying because the linker does not always load a node that is fresh from the load file.

The linker gives each symbol that has an overlay reference an **overlay number**. It uses this value, which is zero or more, to construct the overlay supervisor entry label associated with that symbol. This label is of the form 'OVLYnnnn', where nnnn is the overlay number. You should not use symbols with this format elsewhere.

The linker gathers together all program sections with the same section name. It does this so that it can then load them together in memory.

5.4 Error Codes and Messages

These errors should be rare. If they do occur, the error is probably in the compiler and not in your program. However, you should first check to see that you sent the linker a proper program (for example, an input program must have an introductory program unit that tells the linker to expect a program).

Invalid Object Modules

2 Invalid use of overlay symbol
3 Invalid use of symbol
4 Invalid use of common
5 Invalid use of overlay reference
6 Non-zero overlay reference
7 Invalid external block relocation
8 Invalid bss relocation
9 Invalid program unit relocation
10 Bad offset during 32 bit relocation
11 Bad offset during 16/8 bit relocation
12 Bad offset with 32 bit reference
13 Bad offset with 16/8 bit reference
14 Unexpected end of file
15 Hunk.end missing
16 Invalid termination of file
17 Premature termination of file
18 Premature termination of file

Internal Errors

19 Invalid type in hunk list
20 Internal error during library scan
21 Invalid argument freevector
22 Symbol not defined in second pass

Chapter 6: The System Debugger - DEBUG

This chapter describes the use of the Tripos system debugger. This is a resident task that can be used to inspect and alter store, to set breakpoints, and to single step through a program.

Table of Contents

6.1 Debugging

The DEBUG task in Tripos lets you monitor and modify the code or data of any other task in the system, or of the kernel or device drivers. It has facilities for debugging user programs, and also for handling aborts, whether they occur in a task which is regarded as a user program, or one of the standard system tasks.

DEBUG can work in one of two modes: as a Tripos task communicating with the user via the console handler in the normal way, or in standalone mode with machine interrupts turned off, driving the console keyboard and output device directly. You enter DEBUG Task mode by pressing CTRL-P until you select Task 2. Each time you press CTRL-P, you select the next available task (process); if you find that you have selected a task that you do not want, then you can press CTRL-P again to select the next task, and so on. You enter DEBUG standalone mode while handling an abort or breakpoint; this means that the normal action of Tripos is suspended until you give a C or H command.

Aborts are the general name given to exceptional conditions in Tripos. They may arise from an exception or TRAP instruction; in particular the Tripos routine ABORT causes a TRAP 1 instruction and hence an entry to DEBUG in standalone mode. Many internal Tripos routines call ABORT if something unexpected happens - for example, the library routine SENDPKT aborts if a packet other than the one that it was expecting returns. In this case the argument to ABORT is passed to DEBUG and printed out as part of the abort message. A value is also passed when an exception occurs - for example, a bus error (normally an invalid pointer).

When you enter DEBUG in the standalone mode, DEBUG displays a standard message of the form:

```
!! ABORT Tn: rc message
```

where n is the task number of the aborting task, or 0 if the idle task was running (normally an abort in an interrupt handler), or -1 if a section of the machine code kernel was running. The reason for the abort is given

by rc (return code), which is either the argument passed to ABORT or a number specific to the exception condition. The Tripos command FAULT rc gives more information about the meaning of the numbers.

The DEBUG task may be entered in normal, task mode by selecting task 2 with CTRL-P. In this case, you may use the RUBOUT key (DEL or BACKSPACE) and any console handler escapes; input is not transmitted to DEBUG until ESC or RETURN is pressed. In standalone mode commands are entered at once and RUBOUT discards the current command and displays '??'.

6.2 Examining Store

Store can be examined in a number of ways. The general form for a location in memory is a letter followed by a number. Absolute store locations can be examined by using the command letter 'A' or 'a' (all command letters can be entered in upper or lower case). The number following refers to the BCPL memory address. Thus

```
A256
```

refers to the store location with the BCPL address 256. As BCPL addresses are four times smaller than machine addresses, the actual byte address of that location is 1024. Byte addresses can be specified instead, if required, with the command letter 'M'. Thus

```
M1024
```

refers to the same location.

Hexadecimal notation can be used if required; in this case the hexadecimal number must be prefaced by the symbol '#'. This notation can be used anywhere a number is expected, so the store location above can also be referenced with the syntax

```
A #100
```

or

```
M#400
```

Notice that spaces between the command letter and the argument are optional.

The above mechanism allows a store location to be specified, but does not actually print out the data stored there. It merely causes DEBUG to remember the value specified as the 'current expression'. The value of the current expression can be examined at any time with the command '='. This command displays the value of the current expression, if one exists; otherwise it gives an error. DEBUG indicates all errors by reflecting two question marks: '??'.

It is common to set the current location and examine the contents as a sequence of commands on a single line, but this is not compulsory. For example,

```
A256 =
```

sets the current expression to the contents of the location with BCPL address 256, and then displays the value out. Other commands (which do not alter the value of the current expression) may be inserted between the setting and examining as follows:

```
A256
 ... some other command
=
```

It is common to want to look at a number of store locations, and this can be done with the T command. The T command takes as an argument a number that indicates the number of locations to be typed out. The command starts at the current expression and does not alter the current expression. Thus

```
A256 T20
```

types the contents of location A256, A257 and so on up to A275. As the current location is not altered, you can also give the commands:

```
A256
T20
T10
=
```

This sequence of commands sets the current location to A256, types 20 locations from A256, types 10 locations from A256, and then types the contents of A256.

DEBUG knows a great deal about the structure of store as manipulated by Tripos. It checks to see that the location referenced through A and M commands belongs inside the memory space used by Tripos. This is done to ensure that Bus Errors are not accidentally generated, but in some cases it is useful to access memory addresses outside the range of actual memory used by Tripos. In particular, input/output control and status ports appear in the memory address space of the 68000, and access may be required to these. In this case the command Y is used, which behaves just like A except that the check on the value following Y is not made.

Note: As no checks are made, you must use the Y command with care!

One of the most useful store locations known to DEBUG is the global vector of the current task. The current task is set to the one that aborted in standalone mode, and, initially, to task 1 in task mode. It can be altered by means of the S command to select a different task.

WARNING: An error occurs if an attempt is made to select a task that does not exist.

Global vector locations are referenced by means of the 'G' command. Thus

```
S3 G0 T20
```

selects task 3, sets the current expression to the start of the global vector, and then displays the values of the first 20 globals. Similarly, you might issue the commands:

```
Sl G81 =
```

which print the value of global 81 of task 1.

One of the central data structures in Tripos is the Task Control Block - see the *Tripos Technical Reference Manual* for a full description of this. Locations within the TCB for the current task can be accessed by means of the W command in a similar fashion to the G command.

The register set for a particular task are dumped when a task aborts, and so in standalone mode the command R refers to the memory where this register dump has been kept. The registers are stored D0-D7, A0-A7, SR, PC. The status register (SR) is saved in a long-word location even though it is only 2 bytes long. Thus D0 can be referenced by R0, A0 by R8, and the Program Counter (PC) by R17. Because it is often useful to look at all of the registers, another command is available that simply displays all the registers in suitable form; this is ':' (colon). Colon (:) always displays the registers in hexadecimal.

6.3 Updating Store

As explained above, the = command types the value of the current expression. Store locations can also be examined with the '/' (slash) command. This command not only displays the value of the current expression, but also opens it ready for updating. The location remains open until a RETURN is pressed; this means that in task mode the command '/' immediately followed by a RETURN simply displays the value stored in the location, opens it, and then closes it immediately. In standalone mode DEBUG recognizes the '/' command at once, displays the current value, and waits with the location opened for the next command.

Only locations that are open may be updated. To update an opened location, you use the U (Update) command. U takes a new number as an argument; it then overwrites the old value with this number. Because the location is closed when you press RETURN, the U command must follow the '/' command; in standalone mode the old value is printed once the location is opened, while in task mode the value is printed after the update has been made (but the previous value is printed). Thus

```
A256 / U123
```

opens location A256 and alters the value to 123. To close a location, you can press RETURN or give a DEBUG command other than U, =, or $ (see below).

The next location can be opened with the N (Next) command. This acts just as if the location following the one from which the current expression was obtained was opened by the '/' command. Thus

```
A256 / U123 N U456
```

replaces A256 with the value 123, and then opens location A257 and replaces that with the value 456.

BCPL programs often contain pointers to other parts of store. DEBUG knows about this, and the I (Indirect) command can be used to perform indirection on the current expression, just like the ! operator in BCPL. Thus

```
A256 I
```

takes the value stored at location A256 and uses this as the next address to open. This means that if A256 contained 100, then after this command line was executed the location A100 would be opened, and could be updated (until RETURN is pressed, that is).

In a similar fashion, you can also use the J command to perform indirection, but in this case the value used as the pointer is assumed to be a byte address. For example,

```
A256 J
```

takes the value stored at A256 (for instance, 100) and opens the location referenced by that as a byte address (M100 or A25).

Notice that registers can be altered by opening the register dump using the R (Register) command; in this case the register is reloaded when normal execution continues.

WARNING: Updating any memory location is potentially dangerous.

6.4 Printing Styles

To set the style in which values are typed, you use the '$' command, which takes a letter as an argument. If the style is altered while a location is open or immediately following an = command, the value is reprinted in the new style but is not changed permanently. In all other cases the change is made for good.

The default style is $F, which displays data as follows: if the value appears to be a machine pointer to a BCPL-style function entry point, the name of the function is printed. Thus

```
G81 =
```

displays the name of the function held as global 81 ('loadseg'). Function names are truncated to 7 characters and may have been omitted with a code generator option. If the value in question does not appear to be a function, it is printed in decimal if it is a small number, and in hexadecimal if it is large.

$X sets the style to always print values in hexadecimal, $D always prints values in decimal and $O prints values in octal. Characters can be printed when style $C is selected. This style should be used with care in standalone mode: if the characters include control codes, bizarre effects may be produced on intelligent terminals. Finally, the style $S attempts to print values as BCPL strings. In this case values are examined and if they are valid pointers to store the pointer is assumed to be a string. The first byte at the destination is used as a count and subsequent characters are printed out. Invalid pointers are printed in decimal.

6.5 Expressions

Up to now it has been assumed that the value of the current expression is the content of a store location obtained with commands like A. In fact the current expression can be just a number. For example,

```
123 =
```

sets the current expression to the value 123 and then displays 123. More usefully, you might type:

```
#A05C =
```

to translate hexadecimal to decimal (assuming the current style was $F or $D). Setting the style to hexadecimal or octal will convert in the other direction.

You can also enter the ASCII value of a character by preceding it with a single quote ('). Thus, to find the ASCII representation of the letter A, you could type

```
'A =
```

which sets the current expression to the required value and then prints it.

It has also been assumed so far that all numbers must be entered as a single decimal number, or a hexadecimal number preceded by '#'. In fact the current expression can be set to just that - an expression that will be evaluated. The expression consists of a numbers in either style, or the values of store locations, linked together with a number of operators. For example,

```
A123 + 56 =
```

prints the contents of A123 added to 56.

The operators are as follows:

```
+       Addition
-       Subtraction
*       Multiplication
%       Division
?       Integer remainder
<       Shift places left
>       Shift places right
&       Logical AND
|       Logical OR
!       Indirection (as in BCPL)
```

These operators can be combined as required, with brackets to indicate precedence.

If one of these operators is found as a command, it acts on the current expression. For example,

```
1 + 2 + 3 =
+ 1 =
```

first evaluates 1+2+3, sets the current expression to 6, and prints this out. The command +1 then sets the current expression to its old value plus 1, and prints this out, giving 7.

The same syntax for an expression can also be used wherever a number would be valid, but in this case the expression must be enclosed in brackets. So

```
A (250+6)
```

sets the current expression to the value of A256.

6.6 Continuing from Aborts

An abort causes entry into standalone DEBUG, and suspends the normal action of Tripos. The cause may be examined by any of the DEBUG commands, but eventually you will want to continue with Tripos. To do this, you can use the C (Continue) command, which asks Tripos to continue. If the cause of the abort was a program calling the routine ABORT, then the routine returns and the execution continues.

For other aborts typing C will not work (for example, after a Bus Error the same instruction is tried over and over again). In this case there are three options. The first is the most drastic, and involves typing Z or pressing the reset button. They have the same effect and Tripos must be restarted. This may be the only option open if a program has gone wild and overwritten store.

For less serious cases there are two other alternatives. Typing H holds the current task and allows others to continue. If you have a spare CLI, you can continue working from that, while the other task remains held. The task can be released by entering DEBUG again, making it the current task via the S command, and typing C.

Holding a task is useful if it is looping - the debug task and console handler run at higher priorities than any CLI and so you can always switch to the debug task and hold the offending CLI task. Generally, you need a spare CLI if you have held task 1, and the NEWCLI command can be used to do this. If you have not yet made a new CLI then a facility within DEBUG can be used.

The DEBUG '\' (backslash) command accepts a line of text that you would normally type at a CLI. DEBUG then passes the text over to the RUN command which executes it. For example, typing

```
\newcli
```

to DEBUG is the same as typing RUN NEWCLI to a CLI. Of course, any valid command can follow the '\', but asking for a new CLI is often the most useful. The '\' command can only be used when DEBUG is in task mode, and cannot be used in standalone mode. If a task aborts and you wish to hold it and carry on working on a new CLI, but have not got a

spare one, you should proceed as follows.

Type H to hold the current task. Other tasks now run normally, including the console handler. Press CTRL-P to select DEBUG in task mode. (CTRL-P cycles through the available tasks, so you may need to press it several times before you get task 2.) Now create a new CLI by typing \newcli, and select the new CLI with CTRL-P.

The final way out of an abort is to type K. This command calls the KILLTASK primitive, which causes the command in error to execute the BCPL routine TIDYUP. The standard TIDYUP routine attempts to clear up the world, but cannot close files that are open but not selected, nor can it release space allocated by calls to GetMem. If you have written your own TIDYUP command, then this is called and will, hopefully, return all the resources it was using.

6.7 Breakpoints and Tracing

Breakpoints may be set with the B (Breakpoints) command, which should be followed by an argument in the range 1-9 to indicate the number of the breakpoint to be set. The breakpoint is set at the value of the current expression. For example,

```
G81 B1
```

sets breakpoint 1 at the value of global 81, which would normally be a routine entry point. The value of the current expression is taken to be the byte address of the place where the breakpoint is to be set.

The values of currently set breakpoints can be inspected by means of the command

```
B0
```

A breakpoint can be deleted by setting the current location to zero and then specifying the breakpoint; for example:

```
0 B1
```

After a breakpoint has been found, the program may be continued by typing 'C', which causes one further instruction to be executed and control returned to DEBUG. A further 'C' continues execution. Alternatively, typing a period (.) causes the program to be traced an instruction at a time, using the hardware trace facility of the 68000.

6.8 Disassembly

The Tripos system debugger contains a disassembler. The D (Disassemble) command may be used to obtain 20 lines of program disassembly. The disassembly starts at the 'current disassembly location'. This is either set by quoting a byte address after the D command, or is picked up from the previous value. After disassembly, DEBUG updates the current disassembly location to the location of the last instruction it encountered. After an abort or breakpoint, however, it sets the current disassembly location to the program counter of the offending instruction. No output is produced if an attempt is made to start disassembly at an odd byte address.

When a breakpoint or trace exception is encountered, the current instruction is disassembled and printed out after the !! message.

The registers may be examined by means of the R0 T17 construction; however, the ':' command displays all the registers in hexadecimal, along with their names. This is useful when tracing through code with the '.' command.

6.9 Backtrace

The Tripos system debugger also knows about the structure of BCPL stacks used under Tripos, and these can be inspected by means of the E (Examine) command. The E command takes a letter which indicates the action to be performed. The simplest action is to ask for a complete backtrace of the stack; this is done by EB. In standalone mode there is no way to stop this display rushing off the top of the screen if it is too long.

The other way of performing a backtrace through the stack is interactively. The normal command to enter interactive mode is ET, although any of the others listed below may also be used. Once in interactive mode a special set of subcommands are available that alter the stack display; any of these can also be requested from the normal DEBUG command level by preceding them by the E command. The interactive mode is terminated by typing B (which gives the non-interactive backtrace) or by pressing RETURN.

If the interactive command sequence is left at a particular stack frame, you can use the command letter L to refer to the local variables at that stack frame level. As with other command letters, you follow L with a number that indicates the local variable in question. Thus, once you had selected a suitable stack frame, you could type:

```
L0 T20
```

to print the value of the first 20 local variables. As with all BCPL implementations, these values are, in fact, firstly the arguments to the routine, then the local variables or contents of vectors. There is no indication if the actual local variables have in fact been exhausted, so that L99 in one stack frame may, in fact, refer to L2 in a higher frame.

The interactive commands are as follows. Except for B they all change the values obtained by the L command.

B	do a non-interactive backtrace.
L	set the stack pointer to the current value.
S	set the stack base to the current value.
N	go to next coroutine stack.

T	go to the top of the outermost active coroutine stack; the stack base is obtained from the global vector and the stack pointer from the registers if available, otherwise from the value saved in the TCB at the last interrupt or kernel call.
U	go up to the top of the current stack.
D	go down one level on the current stack, or to the top of the next coroutine stack; this generates the next line of the backtrace.
V	verify the current stack level.

When a new stack pointer or stack base is selected it is verified by typing the appropiate line of backtrace output. This contains the name of the function, the base of the stack frame and the first few local variables.

6.10 Miscellaneous Commands

The 'X' command executes a BCPL callable function, using DEBUG's global vector and stack. The current expression should be the function to be called, as for the 'B' command. It may be followed by up to 4 arguments, separated by spaces. The current expression is set to the result.

There are ten user variables available called V0 to V9. These may be updated and examined in the same way as any other store location, and can be used whenever any other value would be valid.

The O command can be used to specify an offset. This is used when printing out values - the locations are specified in absolute terms and also relative to the offset if this is non-zero. The value specified is a byte offset. Once an offset has been set up, the value of the offset is added to the value given after the M and D commands. This means that once a program is loaded an offset can be specified that is the start of the program in memory. Subsequent references to store can be made by specifying the relative address after M and D commands; the actual base address offset will be added in by DEBUG.

Quick Reference Card

Location names:

A<integer>	Absolute store location <integer> as BCPL address
G<integer>	Global variable <integer>
L<integer>	Local variable <integer>
M<integer>	Absolute store location <integer> as byte address
R<integer>	Register <integer>
V<integer>	Variable <integer>
W<integer>	TCB location <integer>
Y<integer>	absolute store location <integer> (suppress checks)

Commands:

B<integer>	set/delete Breakpoint or list all
C	Continue/release task
D<integer>	Disassemble from byte address <integer>
E	trace Environment
EB	non-interactive Backtrace
ED	Down one level
EL	set stack level to current value
EN	Next coroutine
ES	set Stack base to current value
ET	Top of stack
EU	Up to top of current coroutine
EV	Verify current level
H	Hold current task
I	BCPL Indirection
J	byte indirection
N	Next location
O<integer>	set Offset to be used with M command
S<integer>	Select task
T<integer>	Type contents of <integer> locations
U<e>	Update current location with <e>
X...	eXecute function (up to 4 <integer>s as args)
Z	enter bootstrap and restart

$	set style
$C	characters
$D	decimal
$F	function
$O	octal
$X	hexadecimal
/	open current location
\<text>	Execute <text> as a command
=	type current value
.	Trace one instruction
:	display registers

Chapter 7: Full Screen Support

This chapter describes the full screen support available under Tripos. In particular, it describes the use of the VDU routine.

Table of Contents

7.1 Introduction

Tripos offers a number of commands that use specific actions from a reasonably intelligent VDU. Unfortunately the control codes for these actions can differ from terminal to terminal.

In order to provide support for a number of different terminals, Tripos allows the user to specify which terminal is to be used. The command VDU <name> sets up the terminal as <name> type. If your terminal is not one of those already supported, turn to Chapter 4, "Installation," of the *Tripos Technical Reference Manual* for instructions on how to install it.

The VDU command works by reading a file called DEVS:VDU and constructing, from the specification found there, a section of interpreted code. This code is stored in the console task associated with the CLI.

7.2 VDU

The VDU DOS call has the following format:

```
Res = VDU( code, id, &row, &col )
```

Before this routine can be used, however, the system must be initialized. This is done by calling the VDU function with the vdu.init call. Vdu.init sets the console handler into single character mode and initializes the vdu as required. The console handler must be turned back into normal mode afterwards with the vdu.uninit call.

The value returned from the initialization call is an id which must be passed when making any further calls as the second argument. The first argument is always the code for the required operation, and the other two arguments are the addresses of the variables holding the physical row and column positions. These are updated suitably only if the implementation of a particular feature causes the cursor to be moved from the current physical position and will not be updated if the cursor movement is expected (for example, vdu.left). If the cursor is moved as a

side effect, then it will not be restored; it is the responsibility of the caller to move it back if required. A common programming technique is to maintain the logical cursor position and the physical position. The values whose addresses are passed to the VDU routine represent the physical location. When the cursor is to be displayed for a long period it is placed back at the logical position.

The different calls available are described below.

```
id = VDU ( vdu.init )
```

This call switches the console handler into single character mode, and initializes the VDU (for example, placing into page mode). When in single character mode any key typed at the keyboard is sent immediately, unlike buffered mode when a line is sent only when return is pressed. If the vdu in use has not been defined, the value returned will be zero. Otherwise a screen handle will be returned which must be quoted as the second argument whenever a call is made to the VDU function.

```
VDU( vdu.uninit, id )
```

terminates the use of the VDU in any special way. Unless this call is made the console handler will remain in single character mode, and normal operation of the system will be impossible.

```
length := VDU( vdu.length, id, &row, &col )
width  := VDU( vdu.width,  id, &row, &col )
```

returns the number of lines and the number of characters per line on the screen of the VDU.

```
char := VDU( vdu.getchar, id )
```

returns the next character typed at the keyboard. The system will wait until the character is typed. This call will return the translated character if one has been specified in the VDU definition file.

The characters expected by the ED program and others are as in Table 7-A. All, none or some may be mapped to specialized keys. It is normal for at least the cursor movement keys to be mapped to these control codes. Note that CTRL-S and CTRL-Q are not used as these may be used for flow control.

Action	Code	Control Combination
Insert line	#X01	CTRL-A
Delete line	#X02	CTRL-B
Scroll down	#X04	CTRL-D
Cursor to end screen	#X05	CTRL-E
Flip case	#X06	CTRL-F
Repeat last command	#X07	CTRL-G
Cursor left	#X08	CTRL-H
Tab	#X09	CTRL-I
Cursor down	#X0A	CTRL-J
Cursor up	#X0B	CTRL-K
Return	#X0D	CTRL-M
Delete char	#X0E	CTRL-N
Delete word	#X0F	CTRL-O
Cursor word left	#X12	CTRL-R
Cursor word right	#X14	CTRL-T
Scroll up	#X15	CTRL-U
Verify screen	#X16	CTRL-V
Cursor right	#X18	CTRL-X
Deol	#X19	CTRL-Y
Escape	#X1B	CTRL-[
Cursor to end line	#X1D	CTRL-]

Table 7-A

```
VDU( vdu.setcursor, id, &col, &row )
```

positions the cursor at the col and row specified. The top left hand corner of the screen is position 0,0. Note that the values are passed by reference.

```
VDU( vdu.left,  id, &row, &col)
VDU( vdu.right, id, &row, &col)
VDU( vdu.up,    id, &row, &col)
VDU( vdu.down,  id, &row, &col)
```

moves the cursor one place in the specified directions. This should not be called if it would move the cursor off the edge of the screen.

```
VDU( vdu.dell, id, &row, &col)
```

deletes the current line on the VDU and shuffles up any lines left on the screen to fill the gap. The bottom line is erased.

```
VDU( vdu.insl, id, &row, &col)
```

shuffles down all lines from the current line downwards to make room for a new line. The current line is cleared for input.

```
VDU( vdu.deol, id, &row, &col)
```

deletes all characters from the cursor to the end of the line.

```
res := VDU( vdu.insc, id, &row, &col)
```

shuffles all the characters above and to the right of the cursor one position right. Any character beyond the end of the screen is lost. The character position above the cursor is cleared to a space.

Not all VDUs are capable of this operation. The result returned indicates if it is possible - a TRUE result indicates that the operation has worked, whereas a FALSE result indicates that the VDU cannot support this function.

```
res := VDU( vdu.delc, id, &row, &col)
```

shuffles all characters to the right of the cursor one position left. The last position on the line is cleared to a space. Not all terminals may support this - the result indicates this as detailed above.

```
VDU( vdu.scrollup, id, &row, &col)
```

scrolls all the characters on the screen up one line. The bottom line cleared to blanks. This may be implemented by either a cursor down at the bottom of the screen (often this is the quickest way), or by a delete line at the top of the screen, or in any other way defined in the vdu specification.

```
VDU( vdu.scrolldown, id, &row, &col)
```

scrolls all the characters on the screen down by one line. The top line is cleared to blanks. This may be implemented via a cursor up at the top of the screen, or an insert line at the top of the screen, or in any other way.

```
res := VDU( vdu.highlighton,  id, &row, &col)
res := VDU( vdu.highlightoff, id, &row, &col)
```

sets the screen highlight on or off. The highlight may be extra bright, inverse video, underlined or even nothing at all. It may last for the rest of the line or the rest of the screen (it should always be explictly turned off at the end of the line). Again the result is true or false depending on whether the terminal is capable of supporting this action.

Chapter 8: Floating Point

This chapter describes the implementation of floating point in Tripos.

Table of Contents

8.1 Floating Point Format

Floating point calculations can be single or double precision. The formats of both are described below.

8.1.1 Single Precision

The format for single precision floating point can be expressed as follows:

```
Bit positions                    31-0
 31   30      23   22              0
-----------------------------------
| S | E.......E | M.............M | 1 Long word
-----------------------------------
```

Where

S - Sign bit (0 - positive, 1 - negative)
E - Binary exponent (127 excess)
M - Normalised mantissa (1.MMMMMMMMM)

8.1.2 Double Precision

The format for double precision floating point can be expressed as follows:

```
Bit positions                    63-0
 63   62      52   51              0
-----------------------------------
| S | E.......E | M.............M | 2 Long words
-----------------------------------
```

Where

S - Sign bit (0 - positive, 1 - negative)
E - Binary exponent (103 excess)
M - Normalised mantissa (1.MMMMMMMMM)

8.2 Floating Point Calling Sequence

The calling sequences are different for single and double precision; both are included below.

8.2.1 Single Precision

The single-precision floating point calling sequence is as follows:

```
MOVE.L    <num1>,D1
MOVE.L    <num2>,D2
JSR       FPop
```

On return, D1 holds D1 op D2.

8.2.2 Double Precision

The double-precision floating point calling sequence is as follows:

```
MOVE.L    #num1,D1
MOVE.L    #num2,D2
JSR       DFPop

num1: DS.L      2
num2: DS.L      2
```

The operation performed is (D1) op (D2). The registers remain unchanged.

8.3 Condition Codes

The following condition codes are set:

X - Undefined
N - If result is < 0
Z - If result is 0
V - If underflow or overflow
C - If overflow

8.4 Functions

The following functions are provided:

```
FPADD       D1 + D2 -> D1
FPSUB       D1 - D2 -> D1
FPMUL       D1 * D2 -> D1
FPDIV       D1 / D2 -> D1
FPTST       Test D1 (set condition codes)
FPNEG       -D1 -> D1
FPCMP       D1 - D2 (set condition codes)

DFPADD      (D1) + (D2) -> (D1)
DFPSUB      (D1) - (D2) -> (D1)
DFPMUL      (D1) * (D2) -> (D1)
DFPDIV      (D1) / (D2) -> (D1)
DFPTST      Test (D1) (set condition codes)
DFPNEG      -(D1) -> (D1)
DFPCMP      (D1) - (D2) (set condition codes)
```

Tripos Technical Reference Manual

This manual refers to Issue 6, September 1986

Printed in the U.K

DISCLAIMER

THIS PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT THE DEVELOPER OR METACOMCO PLC OR ITS AFFILIATED DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, METACOMCO PLC OR ITS AFFILIATED COMPANIES DO NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND RESULTS SOLELY AT YOUR OWN RISK. IN NO EVENT WILL METACOMCO PLC OR ITS AFFILIATED COMPANIES BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Tripos Technical Reference Manual

Chapter 1: Filing System

Chapter 2: Tripos Binary File Structure

Chapter 3: Tripos Data Structures

Chapter 4: Installing Tripos

Issue 5 (May 1986)

Chapter 1: The Filing System

This chapter describes the Tripos filing system. It includes information on how to patch a disk corrupted by hardware errors.

Table of Contents

1.1 Tripos File Structure

The Tripos file handler uses a disk that is formatted with blocks of equal size. It provides an indefinitely deep hierarchy of directories, where each directory may contain other directories and files, or just files. The structure is a pure tree - that is, loops are not allowed.

There is sufficient redundancy in the mechanism to allow you to patch together most, if not all, of the contents of a disk after a serious hardware error, for example. To patch the contents of a disk, you can use the DISKED command. For further details on the syntax of DISKED, see Section 1.2, "DISKED - The Disk Editor," later in this chapter. Before you can patch together the contents a disk, you must understand the layout. The subsections below describe the layout of disk pages. Generally, you only use DISKED if DISKDOCTOR fails to fix your disks. There are some things that DISKDOCTOR cannot resolve, and so you should learn how to use DISKED in order to cope when DISKDOCTOR fails. (See Chapter 1 of the *Tripos User's Reference Manual* for a description of DISKDOCTOR.)

1.1.1 Root Block

The root of the tree is the Root Block, which is at a fixed place on the disk. The root is like any other directory, except that it has no parent, and its secondary type is different. Tripos stores the name of the disk volume in the name field of the root block.

Each filing system block contains a checksum, where the sum (ignoring overflow) of all the words in the block is zero.

The figure on the following page describes the layout of the root block.

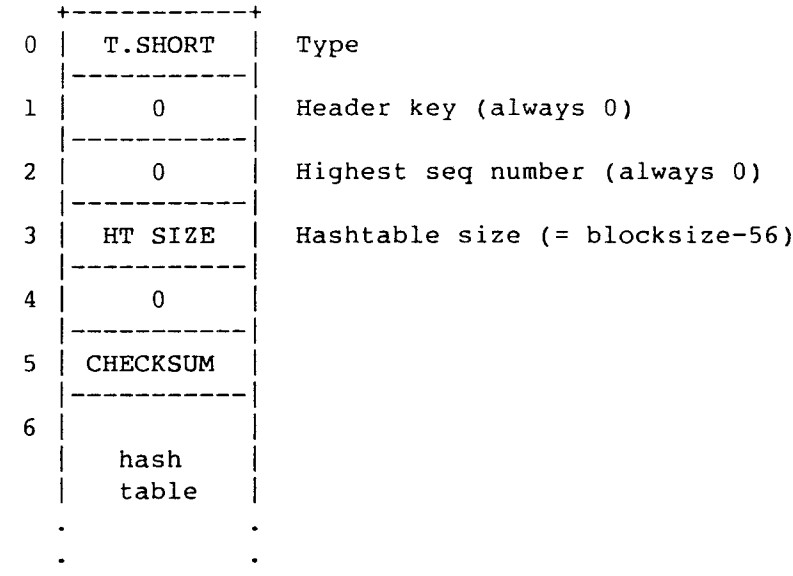


Figure 1-A: Root Block

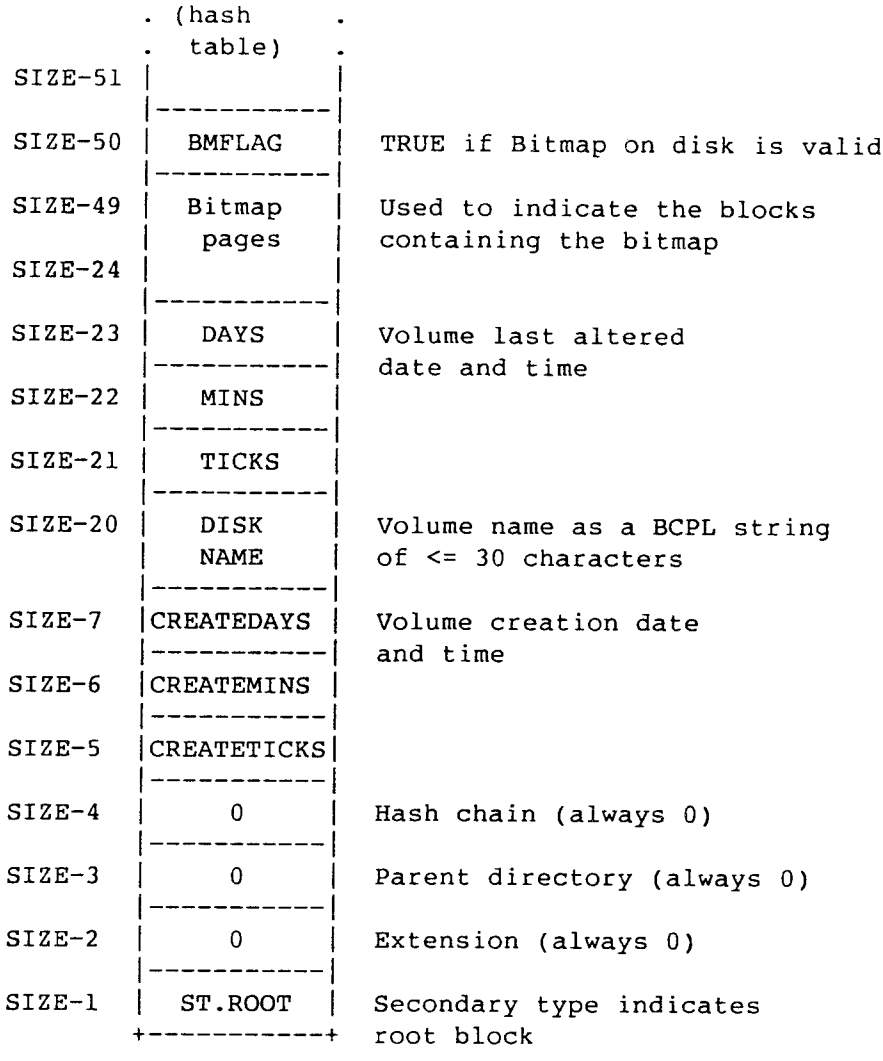


Figure 1-A: Root Block (continued)

1.1.2 User Directory Blocks

The following figure describes the layout of the contents of a user directory block.

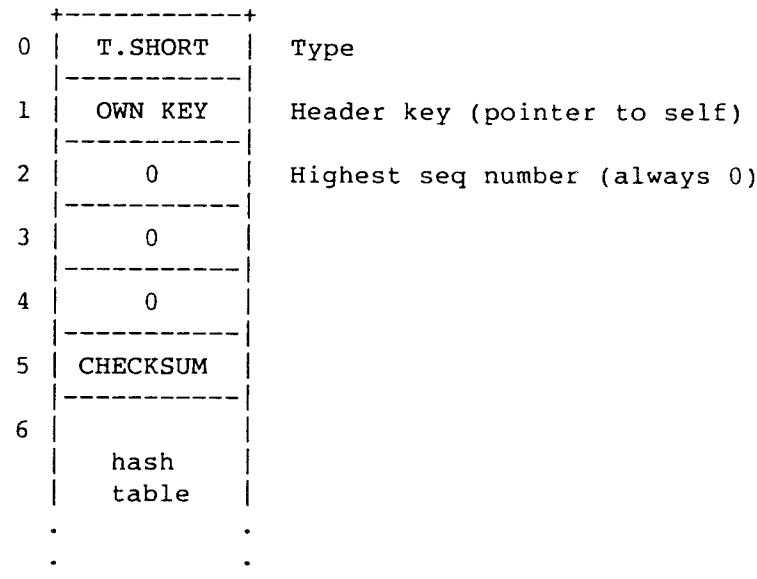


Figure 1-B: User Directory Blocks

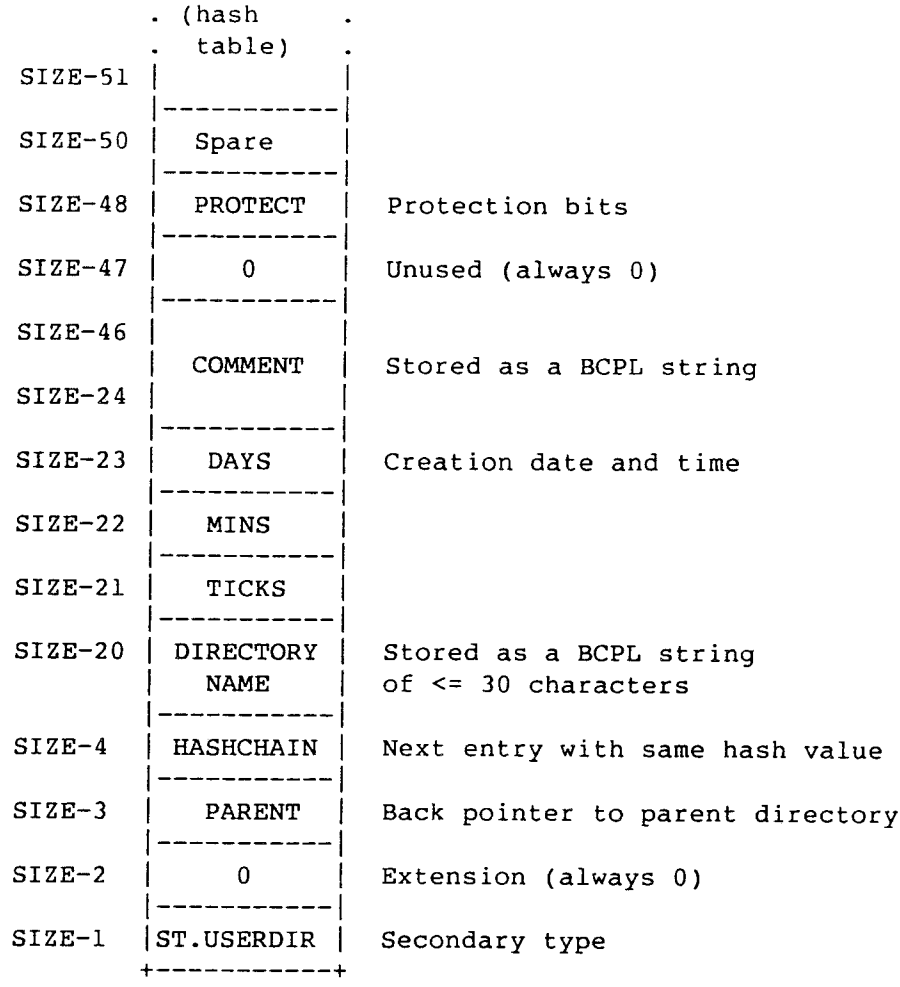


Figure 1-B: User Directory Blocks (continued)

User directory blocks have type T.SHORT and secondary type ST.USERDIRECTORY. The six information words at the start of the block also indicate the block's own key (that is, the block number) as a consistency check and the size of the hash table. The 50 information words at the end of the block contain the date and time of creation, the name of the directory, a pointer to the next file or directory on the hash

chain, and a pointer to the directory above.

To find a file or subdirectory, you must first apply a hash function to its name. This hash function yields an offset from the start of the block to the position in the hash table, which is the key of the first block on a chain linking those with the same hash value (or zero, if there are none). Tripos reads the block with this key and compares the name of the block with the required name. If the names do not match, it reads the next block on the chain, and so on.

1.1.3 File Header Block

The following figure describes the layout of the file header block.

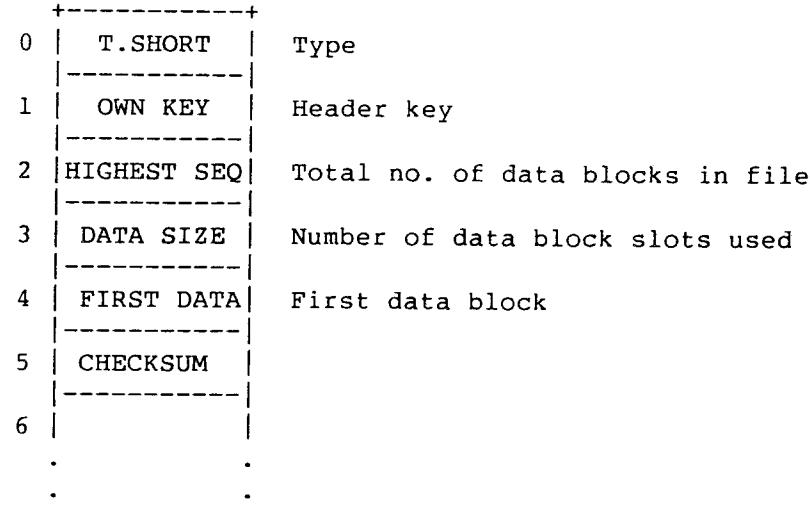


Figure 1-C: File Header Block

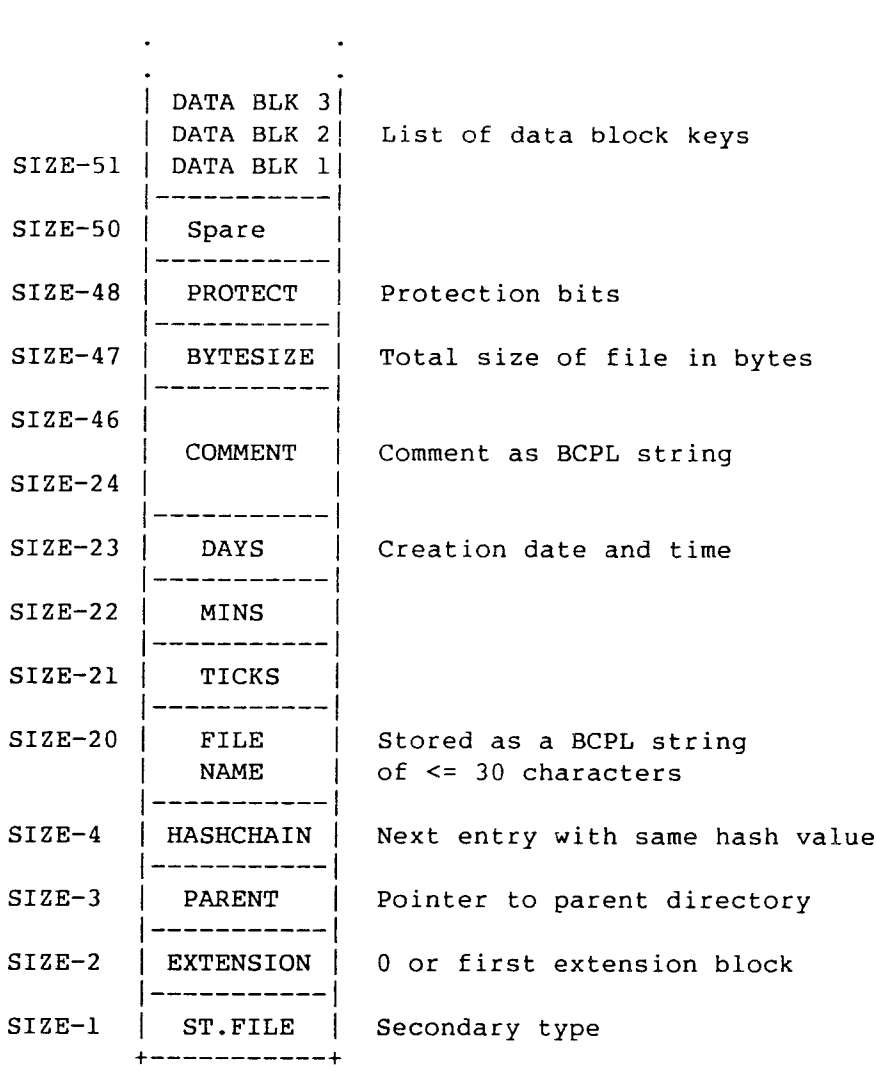


Figure 1-C: File Header Block (continued)

Each file starts with a file header block, which has type T.SHORT and secondary type ST.FILE. The start and end of the block contain name, time, and redundancy information similar to that in a directory block. The body of the file consists of data blocks with sequence numbers from 1

upwards. Tripos stores the addresses of these blocks in consecutive words downwards from offset size-51 in the block. In general, Tripos does not use all the space for this list and the last data block is not full.

1.1.4 File List Block

If there are more blocks in the file than can be specified in the block list, then the EXTENSION field is non-zero and points to another disk block which contains a further data block list. The following figure explains the structure of the file list block.

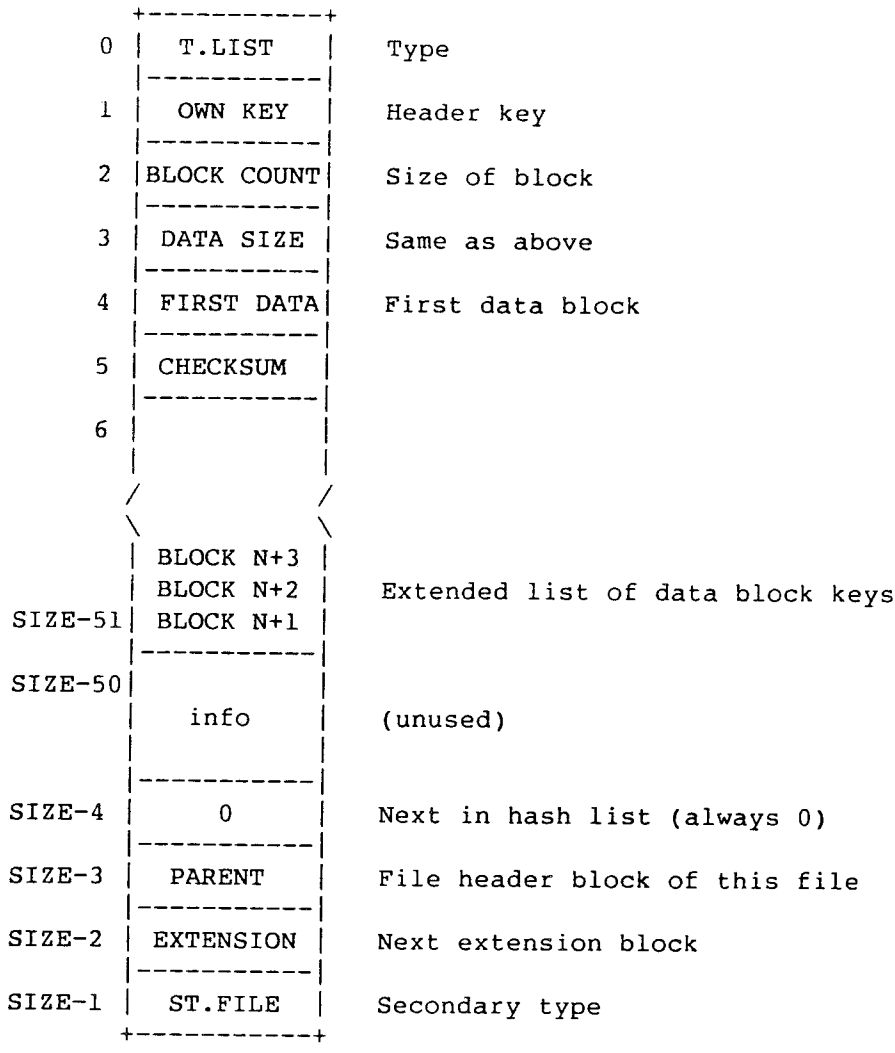


Figure 1-D: File List Block

There are as many file extension blocks as required to list the data blocks that make up the file. The layout of the block is very similar to that of a file header block, except that the type is different and the date and filename fields are not used.

1.1.5 Data Block

The following figure explains the layout of a data block.

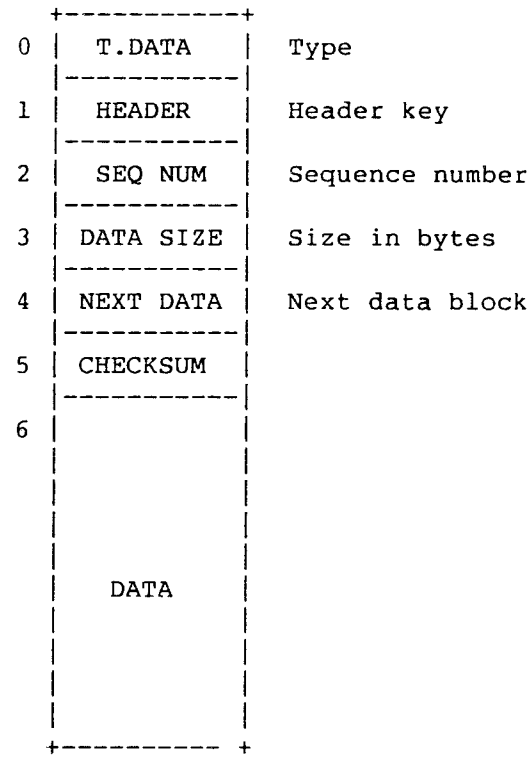


Figure 1-E: Data Block

Data blocks contain only six words of filing system information. These six words refer to the following:

- type (T.DATA)
- pointer to the file header block
- sequence number of the data block
- number of bytes of data
- pointer to the next data block
- checksum

Normally, all data blocks except the last are full (that is, they have a size = blocksize-24). The last data block has a forward pointer of zero.

1.2 DISKED - The Disk Editor

To inspect or patch disk blocks, you can use the Tripos disk editor, DISKED. Because DISKED writes to the disk directly, you should use it with care. Nevertheless, you can use it to good effect in recovering information from a corrupt floppy disk, for example. The template for DISKED is as follows:

```
DISKED "DEV/A"
```

The device parameter must be the name of the disk you wish to edit, followed by a colon (:). This name can refer to a disk drive (for example, DF0:), or to a volume (for example, MyDisk:).

You should only use DISKED with reference to the layout of a Tripos disk. (For a description of the layout, see subsections 1.1.1 to 1.1.5 in the first part of the chapter.) DISKED knows about this structure - for example, the R (Root block) command prints the key of the root block.

Suppose that you give the R command and obtain the key to the root block. You can then give the G (Get block) command followed by this key number to read the block into memory. Once you have used G to get the block, you can use the I (Information) command to print out the information contained in the first and last locations of the block. These locations indicate the type of block, the name, the hash links, and so on. If you then specify a name after an H (Hash) command, DISKED gives you the offset on a directory page that stores as the first key, headers with names that hash to the name you supplied. If you then type the number that DISKED returns followed by a slash (/), DISKED displays the key of that header page. You can then read this with further G commands, and so on.

Consider deleting a file that, due to hardware errors, makes the filing system restart task fail. First, you must locate the directory page that holds the reference to the file. You do this by searching the directory structure from the root block, using the hash codes. Then, you must

locate the slot that references the file - this is either the directory block or a header block on the same hash chain. This slot should contain the key of the file's header block. To set the slot to zero, you type the slot offset, followed by a slash (/) followed by zero (that is, <offset>/0). Then correct the checksum with the K (checKsum) command. You should disable the write protection with X and write back the updated block with P (for Put block) or W (for Windup). You should then change the BM valid flag in the root block from -1 (TRUE) to 0 (FALSE). This causes the disk validator to run when the disk is next inserted. There is no need to do anything else, as the blocks that the file used in error become available once more after the disk validation process has successfully scanned the disk.

DISKED commands are all single characters, sometimes with arguments.

The following is a complete list of the available commands.

Command	Function
B n	Set logical block number base to n
C n	Display n characters from current offset
G [n]	Get block n from disk (default is the current block number)
H name	Calculate hash value of name
I	Display block information
K	Check block checksum (and correct if wrong)
L [lwb upb]	Locate words that match Value under Mask (lwb and upb restrict search)
M n	Set Mask (for L and N commands) to n
N [lwb upb]	Locate words that do not match Value under Mask
P n	Put block in memory to block n on disk (default is the current block number)
R	Display block number of Root Block
Q	Quit (do not write to disk)
S char	Set display Style char = C -> characters S -> string O -> octal X -> hex D -> decimal
T lwb upb	Type range of offsets in block
V n	Set Value for L and N commands
W	Windup (=PQ)
X	Invert write protect state
Y n	Set cYlinder base to n
Z	Zero all words of buffer
number	Set current word offset in block = Display values set in program
/ [n]	Display word at current offset or update value to n
'chars'	Put chars at current offset

Table 1.A: DISKED Commands

To indicate octal or hex, you can start numbers with # or #X (that is, # for octal, #X for hex). You can also include BCPL string escapes (*N and so forth) in strings.

Chapter 2: Binary File Structure

This chapter describes the structure of binary object files under Tripos, as produced by assemblers and compilers. It also describes the format of binary load files, which are produced by the linker and read into memory by the loader.

Table of Contents

2.1 Introduction

This chapter describes the structure of binary object files under Tripos, as produced by assemblers and compilers. It also describes the format of binary load files, which are produced by the linker and read into memory by the loader. The format of load files supports overlaying. Apart from describing the format of load files, this chapter explains the use of common symbols, absolute external references, and program units.

2.1.1 Terminology

Some of the technical terms used in this chapter are explained below.

External References

You can use a name to specify a reference between separate program units. The data structure lets you have a name longer than 16Mbytes, although the linker restricts names to 255 characters. When you link the object files into a single load file, you must ensure that all external references match corresponding external definitions. The external reference may be of size byte, word, or long; external definitions refer to relocatable values or absolute values. Relocatable byte and word references refer to PC-relative address modes and these are entirely handled by the linker. However, if you have a program containing longword relocatable references, relocation may take place when you load the program.

Note that these sizes only refer to the length of the relocation field; it is possible to load a word from a long external address, for example, and the linker makes no attempt to check that you are consistent in your use of externals.

Object File

An assembler or compiler produces a binary image, called an object file. An object file contains one or more program units. It may also contain external references to other object files.

Load File

The linker produces a binary image from a number of object files. This binary image is called a load file. A load file does not contain any unresolved external references.

Program Unit

A program unit is the smallest element the linker can handle. A program unit can contain one or more hunks; object files can contain one or more program units. If the linker finds a suitable external reference within a program unit when it inspects the scanned libraries, it includes the entire program unit in the load file. An assembler usually produces a single program unit from one assembly (containing one or more hunks); a compiler such as FORTRAN produces a program unit for each subroutine, main program, or BLOCK DATA. Hunk numbering starts from zero within each program unit; the only way you can reference other program units is through external references.

Hunks

A hunk consists of a block of code or data, relocation information, and a list of defined or referenced external symbols. Data hunks may specify initialized data or uninitialized data (bss). bss hunks may contain external definitions but no external references nor any values requiring relocation. If you place initialized data blocks in overlays, the linker should not normally alter these data blocks, since it reloads them from disk during the overlay task. Hunks may be named or unnamed, and they may contain a symbol table in order to provide symbolic debugging information. They may also contain a further debugging information for the use of high level language debugging tools. Each hunk within a program unit has a number, starting from zero.

Scanned library

A scanned library consists of object files that contain program units which are only loaded if there are any outstanding external references to them. You may use object files as libraries and provide them as primary input to the linker, in which case the input includes all the program units the object files contain. Note that you may concatenate object files.

Node

A node consists of at least one hunk. An overlaid load file contains a root node, which is resident in memory all the time that the program is running, and a number of overlay nodes which are brought into memory as required.

2.2 Object File Structure

An object file is the output of the assembler or a language translator. To use an object file, you must first resolve all the external references. To do this, you pass the object file through the linker. An object file consists of one or more program units. Each program unit starts with a header and is followed by a series of hunks joined end to end, each of which contains a number of 'blocks' of various types. Each block starts with a longword which defines its type, and this is followed by zero or more additional longwords. Note that each block is always rounded up the nearest longword boundary. The program unit header is also a block with this format.

The format of a program unit is as follows:

- Program unit header block
- Hunks

The basic format of a hunk is as follows:

- Hunk name block
- Relocatable block
- Relocation information block
- External symbol information block
- Symbol table block
- Debug block
- End block

You may omit all these block types, except the end block.

The following subsections describe the format of each of these blocks. The value of the type word appears in decimal and hex after the type name, for example, hunk_unit has the value 999 in decimal and 3E7 in hex.

2.2.1 hunk_unit (999/3E7)

This specifies the start of a program unit. It consists of a type word, followed by the length of the unit name in longwords, followed by the name itself padded to a longword boundary with zeros, if required. In diagramatic form, the format is as follows:

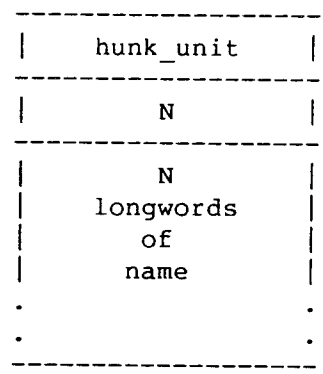


Figure 2-A: Hunk_unit (999/3E7)

2.2.2 hunk_name (1000/3E8)

This defines the name of a hunk. Names are optional; if the linker finds two or more named hunks with the same name, it combines the hunks into a single hunk. Note that 8 or 16 bit program counter relative external references can only be resolved between hunks with the same name. Any external references in a load format file are between different hunks and require 32 bit relocatable references; although, as the loader scatter loads the hunks into memory, you cannot assume that they are within 32K of each other. Note that the length is in longwords and the name block, like all blocks, is rounded up to a longword boundary by padding with zeros. The format is as follows:

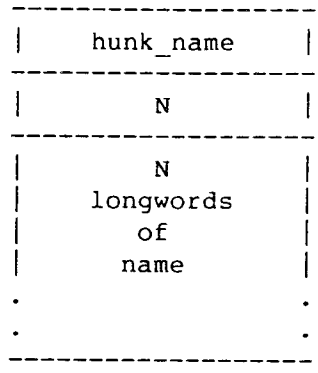


Figure 2-B: Hunk_name (1000/3E8)

2.2.3 hunk_code (1001/3E9)

This defines a block of code that is to be loaded into memory and possibly relocated. Its format is as follows:

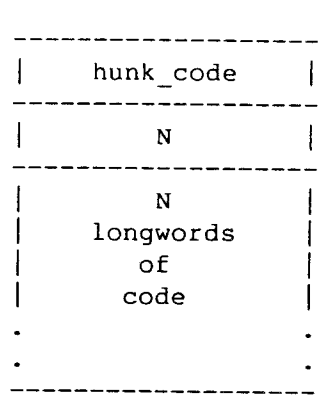


Figure 2-C: Hunk_code (1001/3E9)

2.2.4 hunk_data (1002/3EA)

This defines a block of initialized data which is to be loaded into memory and possibly relocated. The linker should not alter these blocks if they are part of an overlay node, as it may need to reread them from disk during overlay handling. The format is as follows:

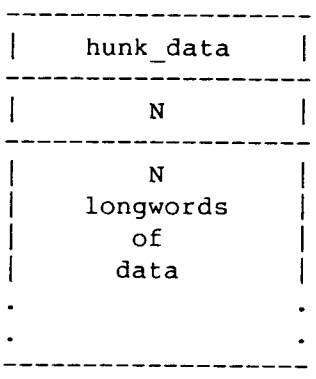


Figure 2-D: Hunk_data (1002/3EA)

2.2.5 hunk_bss (1003/3EB)

This specifies a block of uninitialized workspace which is allocated by the loader. bss blocks are used for such things as stacks and for FORTRAN COMMON blocks. It is not possible to relocate inside a bss block, but symbols can be defined within one. Its format is as follows:

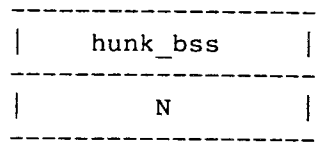


Figure 2-E: Hunk_bss (1003/3EB)

where N is the size of block you require in longwords. The memory used for bss blocks is zeroed by the loader when it is allocated.

The relocatable block within a hunk must be one of hunk_code, hunk_data or hunk_bss.

2.2.6 hunk_reloc32 (1004/3EC)

A hunk_reloc32 block specifies 32 bit relocation that the linker is to perform within the current relocatable block. The relocation information is a reference to a location within the current hunk or any other within the program unit. Each hunk within the unit is numbered, starting from zero. The linker adds the address of the base of the specified hunk to each of the longwords in the preceding relocatable block that the list of offsets indicates. The offset list only includes referenced hunks and a count of zero indicates the end of the list. Its format is as follows:

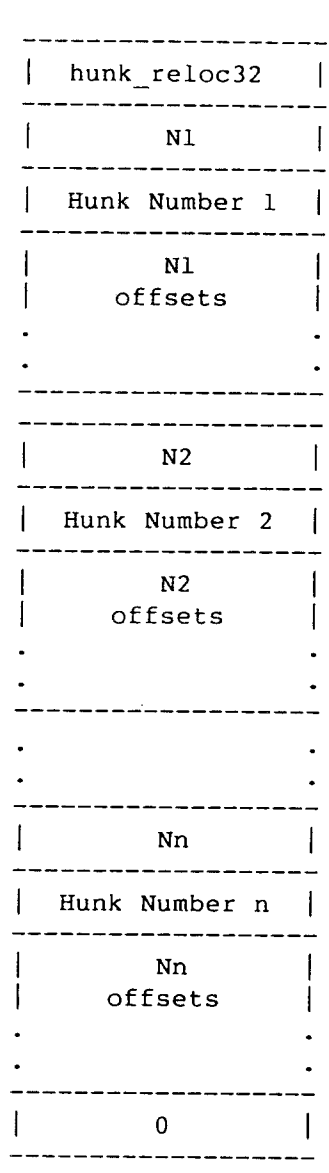


Figure 2-F: Hunk_reloc32 (1004/3EC)

2.2.7 hunk_reloc16 (1005/3ED)

A hunk_reloc16 block specifies 16 bit relocation that the linker should perform within the current relocatable block. The relocation information refers to 16 bit program counter relative references to other hunks in the program unit. The format is the same as hunk_reloc32 blocks. These references must be to hunks with the same name, so that the linker can perform the relocation while it coagulates (that is, gathers together) similarly named hunks.

2.2.8 hunk_reloc8 (1006/3EE)

A hunk_reloc8 block specifies 8 bit relocation that the linker should perform within the current relocatable block. The relocation information refers to 8 bit program counter relative references to other hunks in the program unit. The format is the same as hunk_reloc32 blocks. These references must be to hunks with the same name, so that the linker can perform the relocation while it coagulates similarly named hunks.

2.2.9 hunk_ext (1007/3EF)

This block contains external symbol information. It contains entries both defining symbols and listing references to them. Its format is as follows:

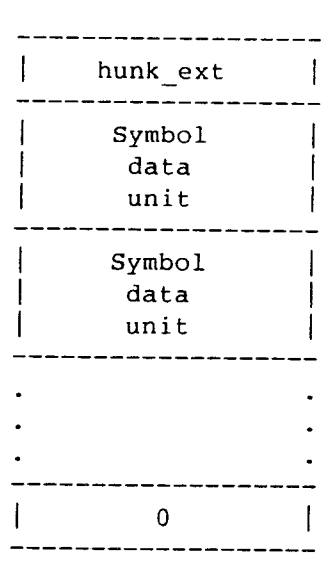


Figure 2-G: Hunk_ext (1007/3EF)

where there is one 'symbol data unit' for each symbol used, and the block ends with a zero word.

Each symbol data unit consists of a type byte, the symbol name length (three bytes), the symbol name itself, and further data. You specify the symbol name length in longwords, and pad the name field to the next longword boundary with zeros.

The type byte specifies whether the symbol is a definition or a reference, etc. Tripos uses values 0-127 for symbol definitions, and 128-255 for references.

At the moment, the values are as follows:

Name	Value	Meaning
ext__symb	0	Symbol table
ext__def	1	Relocatable definition
ext__abs	2	Absolute definition
ext__ref32	129	32 bit reference to symbol
ext__common	130	32 bit reference to COMMON
ext__ref16	131	16 bit reference to symbol
ext__ref8	132	8 bit reference to symbol

Table 2-A: External Symbols

The linker faults all other values. For ext__def there is one data word, the value of the symbol. This is merely the offset of the symbol from the start of the hunk. For ext__abs there is also one data value, which is the absolute value to be added into the code. The linker treats the value for ext__res in the same way as ext__def, except that it assumes the hunk name is the library name and it copies this name through to the load file. The type bytes ext__ref32, ext__ref16 and ext__ref8 are followed by a count and a list of references, again specified as offsets from the start of the hunk.

The type ext__common has the same structure except that it has a COMMON block size before the count. The linker treats symbols specified as common in the following way: if it encounters a definition for a symbol referenced as common, then it uses this value (the only time a definition should arise is in the FORTRAN Block Data case). Otherwise, it allocates suitable bss space using the maximum size you specified for each common symbol reference.

The linker handles external references differently according to the type of the corresponding definition. It adds absolute values to the long, word, or byte field and gives an error if the signed value does not fit. Relocatable 32 bit references have the symbol value added to the field and a relocation record is produced for the loader. 16 and 8 bit references are handled as PC-relative references and may only be made to hunks with the same name so that the hunks are coagulated by the linker before they are loaded. It also possible for PC relative references to fail if

the reference and the definition are too far apart. The symbol data unit formats are as follows:

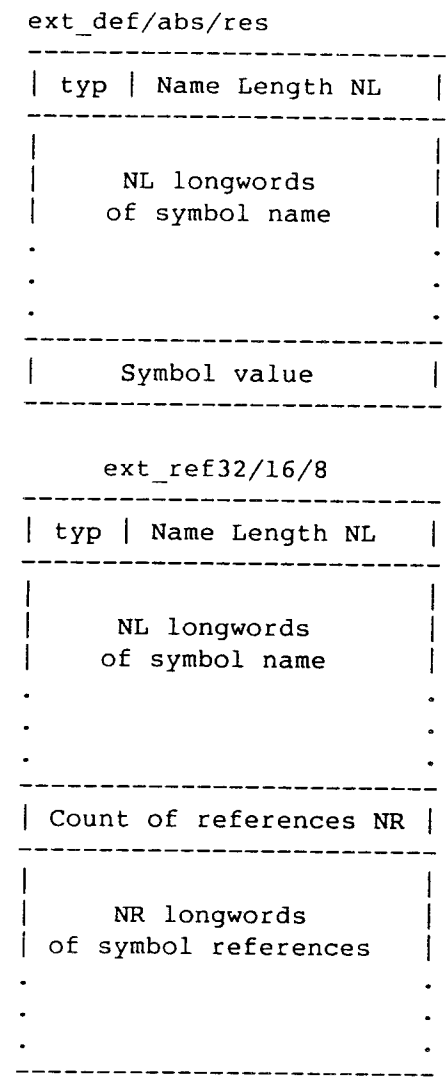
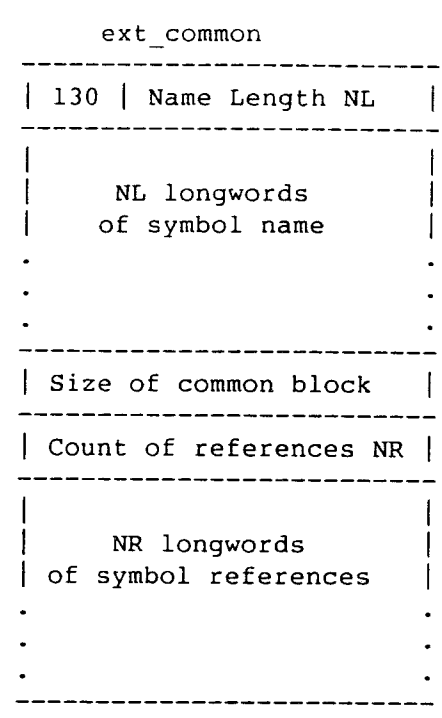


Figure 2-H: Symbol Data Unit



(continuation of Figure 2-H)

2.2.10 hunk_symbol (1008/3F0)

You use this block to attach a symbol table to a hunk so that you can use a symbolic debugger on the code. The linker passes symbol table blocks through attached to the hunk and, if the hunks are coagulated, coagulates the symbol tables. The loader does not load symbol table blocks into memory; when this is required, the debugger is expected to read the load file. The format of the symbol table block is the same as the external symbol information block with symbol table units for each name you use. The type code of zero is used within the symbol data units. The value of the symbol is the offset of the symbol from the start of the hunk. Thus the format is as follows:

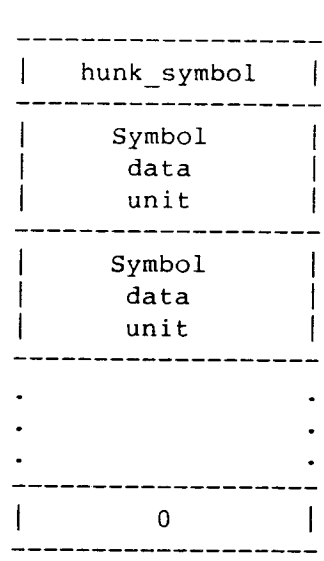


Figure 2-I: Hunk_symbol (1008/3F0)

where each symbol data unit has the following format.

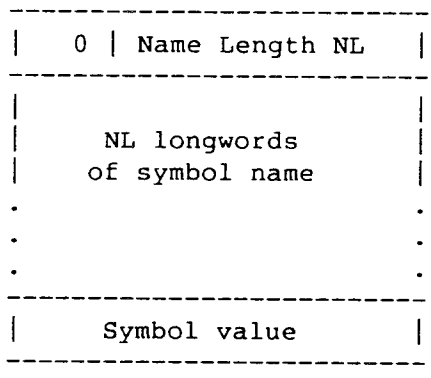


Figure 2-J: Symbol Data Unit

2.2.11 hunk_debug (1009/3F1)

Tripos provides the debug block so that an object file can carry further debugging information. For example, high level language compilers may need to maintain descriptions of data structures for use by high level debuggers. The debug block may hold this information. Tripos does not impose a format on the debug block except that it must start with the hunk_debug longword and be followed by a longword that indicates the size of the block in longwords. Thus the format is as follows:

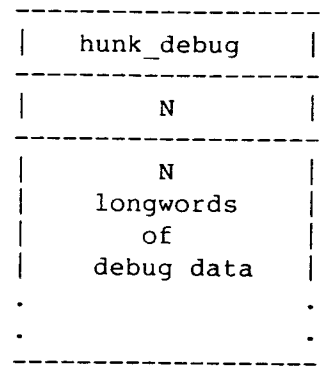


Figure 2-K: Hunk_debug (1009/3F1)

2.2.12 hunk_end (1010/3F2)

This specifies the end of a hunk. It consists of a single longword, hunk_end.

2.3 Load Files

The format of a load file (that is, the output from the linker) is similar to that of an object file. In particular, it consists of a number of hunks with a similar format to those in an object file. The main difference is that the hunks never contain an external symbol information block, as all external symbols have been resolved, and the program unit information is not included. In a simple load file that is not overlaid, the file contains a header block which indicates the total number of hunks in the load file. This block is followed by the hunks, which may be the result of coagulating a number of input hunks if they had the same name. This complete structure is referred to as a node. Load files may also contain overlay information. In this case, an overlay table follows the primary node, and a special break block separates the overlay nodes. Thus the load file structure can be summarized as follows, where the items marked with an asterisk (*) are optional.

- Primary node
- Overlay table block (*)
- Overlay nodes separated by break blocks (*)

The relocation blocks within the hunks are always of type hunk_reloc32, and indicate the relocation to be performed at load time. This includes both the 32 bit relocation specified with hunk_reloc32 blocks in the object file and extra relocation required for the resolution of external symbols.

Each external reference in the object files is handled as follows. The linker searches the primary input for a matching external definition. If it does not find one, it searches the scanned library and includes in the load file the entire program unit where the definition was defined. This may make further external references become outstanding. At the end of the first pass, the linker knows all the external definitions and the total number of hunks that it is going to use. These include the hunks within the load file. On the second pass, the linker patches the longword external references so that they refer to the required offset within the hunk which defines the symbol. It produces an extra entry in the relocation block so that, when the hunks are loaded, it adds to each external reference the base address of the hunk defining the symbol.

Before the loader can make these cross hunk references, it needs to know the number and size of the hunks in the nodes. The header block provides this information, as described below. The load file may also contain overlay information in an overlay table block. Break blocks separate the overlay nodes.

2.3.1 hunk_header (1011/3F3)

This block gives information about the number of hunks that are to be loaded, and the size of each one.

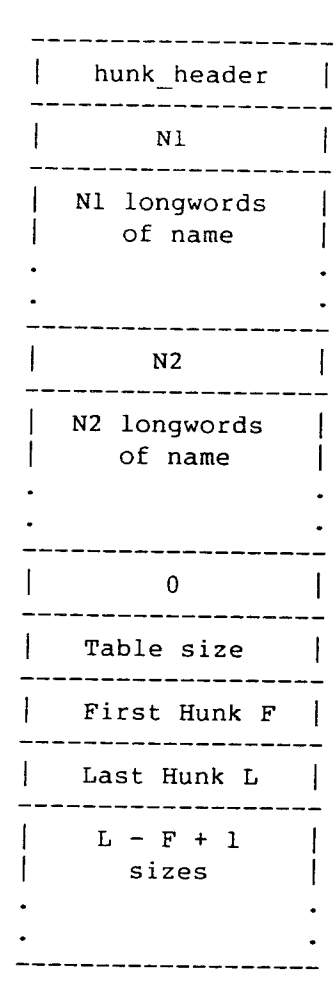


Figure 2-L: Hunk_header (1011/3F3)

The format of the hunk_header is described in Figure 2-L. The first part of the header block contains the names that the loader must open when this node is loaded. Each name consists of a long word indicating the length of the name in longwords and the text name padded to a longword boundary with zeros. The name list ends with a longword of zero. The names are in the order in which the loader is to open them.

When it loads a primary node, the loader allocates a table in memory which it uses to keep track of all the hunks it has loaded. This table must be large enough for all the hunks in the load file, including the hunks in overlays. The next longword in the header block is therefore this table size, which is equal to the maximum hunk number referenced plus one.

The next longword F refers to the first slot in the hunk table the loader should use when loading.

The next longword L refers to the last hunk slot the loader is to load as part of this loader call. The total number of hunks loaded is therefore L - F + 1.

The header block continues with L - F + 1 longwords which indicate the sizes of each hunk which is to be loaded as part of this call. This enables the loader to preallocate the space for the hunks and hence perform the relocation between hunks which is required as they are loaded. One hunk may be the bss hunk with a size given as zero; in this case the loader uses an operating system variable to give the size as described in hunk_bss above.

2.3.2 hunk_overlay (1013/3F5)

The overlay table block indicates to the loader that it is loading an overlaid program, and contains all the data for the overlay table. On encountering it, the loader sets up the table, and returns, leaving the input channel to the load file still open. Its format is as follows:

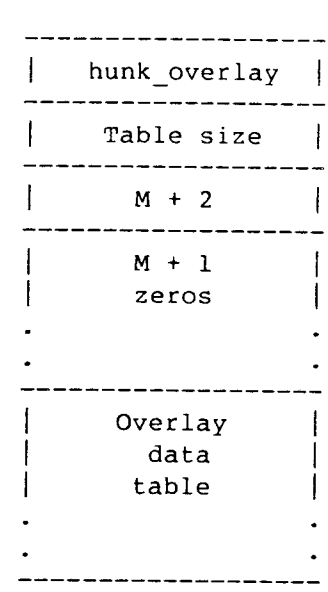


Figure 2-M: Hunk_overlay (1013/3F5)

The first longword is the upper bound of the complete overlay table (in longwords).

M is the maximum level the overlay tree uses with the root level being zero. The next M+1 words form the ordinate table section of the overlay table.

The rest of the block is the overlay data table, a series of eight-word entries, one for each overlay symbol. If O is the maximum overlay number used, then the size of the overlay data table is (O+1)*8, since the first overlay number is zero. So, the overlay table size is equal to (O+1)*8 + M+1.

2.3.3 hunk_break (1014/3F6)

A break block indicates the end of an overlay node. It consists of a single longword, hunk_break.

2.4 Examples

The following simple sections of code show how the linker and loader handle external symbols. For example,

```
                            IDNT    A
                            XREF    BILLY,JOHN
                            XDEF    MARY
* The next longword requires relocation
0000' 0000 0008             DC.L    FRED
0004' 123C 00FF             MOVE.B  #$FF,D1
0008' 7001            FRED  MOVEQ   #1,D0
* External entry point
000A' 4E71            MARY  NOP
000C' 4EB9 0000 0000        JSR     BILLY   Call external
*Now reference the external
0012' 2239 0000 0000        MOVE.L  JOHN,D1
                            END
```

produces the following object file

```
hunk_unit
00000001        Size in longwords
41000000        Name, padded to lword
hunk_code
00000006        Size in longwords
00000008 123C00FF 70014E71 4EB90000 00002239 00000000
hunk_reloc32
00000001        Number in hunk 0
00000000        Hunk 0
00000000        Offset to be relocated
00000000        Zero to mark end
hunk_ext
01000001        XDEF, Size 1 longword
4D415259        MARY
0000000A        Offset of definition
81000001        XREF, Size 1 longword
4A4F484E        JOHN
00000001        Number of references
00000014        Offset of reference
81000002        XREF, Size 2 longwords
42494C4C        BILLY
59000000        (zeros to pad)
00000001        Number of references
0000000E        Offset of reference
00000000        End of external block
hunk_end
```

The matching program to this is as follows:

```
                                IDNT    B
                                XDEF    BILLY,JOHN
                                XREF    MARY
0000' 2A3C  AAAA  AAAA          MOVE.L  #$AAAAAAAA,D5
* External entry point
0006' 4E71              BILLY   NOP
* External entry point
0008' 7201              JOHN    MOVEQ   #1,D1
* Call external reference
000A' 4EF9  0000  0000          JMP     MARY
                                END
```

and the corresponding output code would be

```
hunk_unit
00000001          Size in longwords
42000000          Unit name
hunk_code
00000004          Size in longwords
2A3CAAAA AAAA4E71 72014EF9 00000000
hunk_ext
01000001          XDEF, Size 1 longword
4A4F484E          JOHN
00000008          Offset of definition
01000002          XDEF, Size 2 longwords
42494C4C          BILLY
59000000          (zeros to pad)
00000006          Offset of definition
81000001          XREF, Size 1 longword
4D415259          MARY
00000001          Number of references
0000000C          Offset of reference
00000000          End of external block
hunk_end
```

Once you have passed this through the linker, the load file will have the following format.

```
hunk_header
00000000        No hunk name
00000002        Size of hunk table
00000000        First hunk
00000001        Last hunk
00000006        Size of hunk 0
00000004        Size of hunk 1
hunk_code
00000006        Size of code in longwords
00000008 123C00FF 70014E71 4EB90000 00062239 00000008
hunk_reloc32
00000001        Number in hunk 0
00000000        Hunk 0
00000000        Offset to be relocated
00000002        Number in hunk 1
00000001        Hunk 1
00000014        Offset to be relocated
0000000E        Offset to be relocated
00000000        Zero to mark end
hunk_end
hunk_code
00000004        Size of code in longwords
2A3CAAAA AAAA4E71 72014EF9 0000000A
hunk_reloc32
00000001        Number in hunk 0
00000000        Hunk 0
0000000C        Offset to be relocated
00000000        Zero to mark end
hunk_end
```

When the loader loads this code into memory, it reads the header block and allocates a hunk table of two longwords. It then allocates space by calling an operating system routine and requesting two areas of sizes 6 and 4 longwords respectively. Assuming the two areas it returned were at locations 3000 and 7000, the hunk table would contain 3000 and 7000.

The loader reads the first hunk and places the code at 3000; it then handles relocation. The first item specifies relocation with respect to hunk 0, so it adds 3000 to the longword at offset 0 converting the value stored there from 00000008 to 00003008. The second item specifies

relocation with respect to hunk 1. Although this is not loaded, we know that it will be loaded at location 7000, so this is added to the values stored at 300E and 3014. Note that the linker has already inserted the offsets 00000006 and 00000008 into the references in hunk 0 so that they refer to the correct offset in hunk 1 for the definition. Thus the longwords specifying the external references end up containing the values 00007006 and 00007008, which is the correct place once the second hunk is loaded.

In the same way, the loader loads the second hunk into memory at location 7000 and the relocation information specified alters the longword at 700C from 0000000A (the offset of MARY in the first hunk) to 0000300A (the address of MARY in memory).

Chapter 3: Tripos Data Structures

This chapter describes Tripos data structures in memory and in files. It does not describe the layout of a disk, which is described in Chapter 1.

Table of Contents

3.1 Introduction

This chapter describes the data structures within Tripos, including the format of a task, central shared data structures, and the structure of handler requests.

In addition to normal values such as integers and pointers, Tripos uses BPTR. BPTR is a BCPL pointer, which is a pointer to a longword-aligned memory block divided by 4. So, to read a BPTR in C, you simply shift left by 2. To create a BPTR, you must either use memory obtained via a call to GetMem or a structure on your stack when you know you have only allocated longwords on the stack so far (the initial stack is longword aligned). You should then shift this pointer right by 2 to create the BPTR.

Tripos also has a BSTR, which is a BCPL string. BSTR consists of a BPTR to memory that contains the length of the string in the first byte, and the bytes within the string following.

3.2 Global Data Structure

This section describes the rootnode. The rootnode is the central point from which all the system structures in memory can be found. It is a vector containing pointers to the main chains and tables, and certain other information detailed below. In fact, the location of the rootnode is the only fixed location in memory. All other data structures are obtainable from the rootnode. The value zero is usually used as a null pointer (for example, to mark the end of chains). In the box diagrams, each cell represents one machine word (longword).

3.2.1 BLKLIST

This points to the start of the area from which memory is allocated by GetMem. Fuller details of the free memory layout are given later.

3.2.2 DAYS, MINS and TICKS

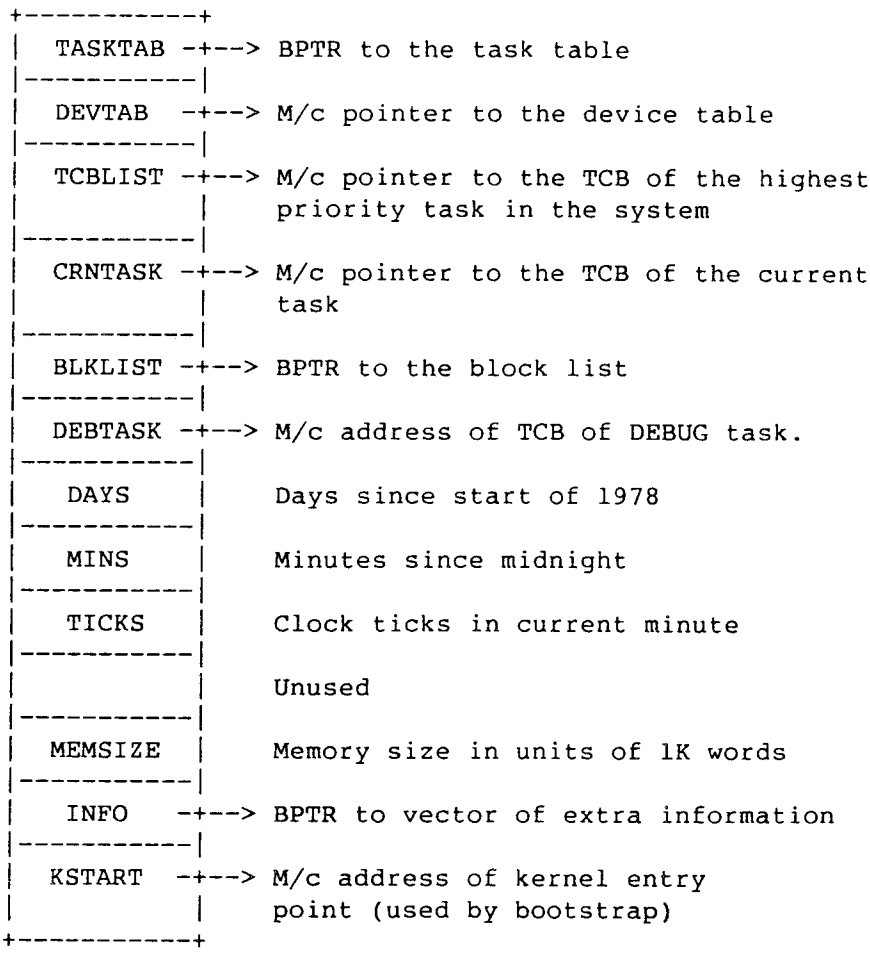
The clock interrupt routine maintains the date and time in these three words. DAYS is the number of days since the start of 1978 (that is, January 1st, 1978 is day 0). MINS is the number of minutes since midnight. TICKS is the number of clock ticks since the last minute boundary. The time is updated at a frequency given by the system constant TICKSPERSECOND.

3.2.3 INFO

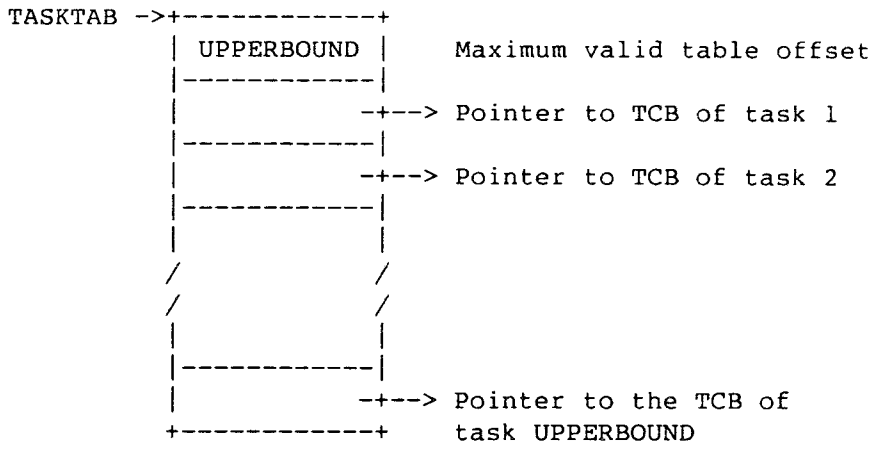
The INFO field points to a substructure used to hold information about device assignments, details of machine type, etc. This substructure may be extended without altering the structure of the rootnode. Fuller details are given later.

3.2.4 KSTART

This value is assembled into the rootnode, and is the means by which the bootstrap finds the kernel entry point after loading the system into memory.

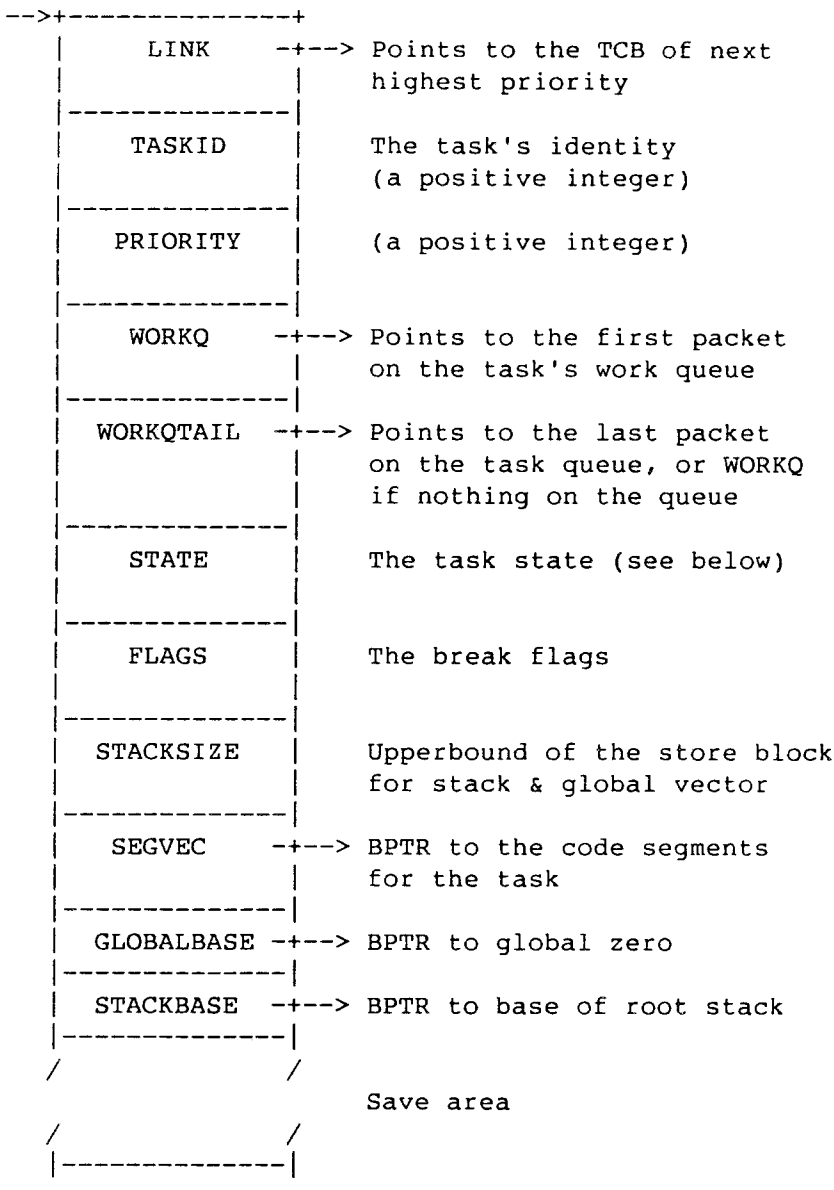

TASKTAB
BPTR to the task table
DEVTAB
M/c pointer to the device table
TCBLIST
M/c pointer to the TCB of the highest priority task in the system
CRNTASK
M/c pointer to the TCB of the current task
BLKLIST
BPTR to the block list
DEBTASK
M/c address of TCB of DEBUG task.
DAYS
Days since start of 1978
MINS
Minutes since midnight
TICKS
Clock ticks in current minute
Unused
MEMSIZE
Memory size in units of 1K words
INFO
BPTR to vector of extra information
KSTART
M/c address of kernel entry point (used by bootstrap)

3.2.5 TASKTAB



This table enables a task control block to be found from its corresponding task id. Unused elements contain zero.

3.2.6 Task Control Block (TCB)



Each task in the system has a task control block (TCB) which contains information relating to the task. There are two parts to a TCB. The first part contains information used by the operating system for controlling the task. The second part contains the save area used to hold the machine registers, program counter, and processor status, when a task suspends itself or is interrupted.

3.2.7 The Task State

This is held in the least significant 4 bits of the state field. All the remaining bits are zero. The significance of each bit is as follows:-

0001: **Packet bit.** If this bit is set, the task has at least one packet on its work queue. If clear, then the work queue is empty.

0010: **Held bit.** This bit is set when the task is in held state. It means that the task will not be selected for running, even though it might otherwise be eligible. Its primary purpose is as a debugging aid.

0100: **Wait bit (dead state).** This is set when the task is waiting for a packet to arrive. The task will not run while its work queue is empty. Note that this bit pattern dictates that the task is dead since a task that has called TaskWait will have also set the interrupted bit (see below).

1000: **Interrupted bit.** When this bit is set, the task has been interrupted. The task will run again when the interrupt service routine is complete, and any higher priority tasks it may have activated are once again held up.

All of the 16 possible bit patterns are valid. This means that the task selector can rapidly decide how to deal with a task, by using the state to index a table of routine addresses.

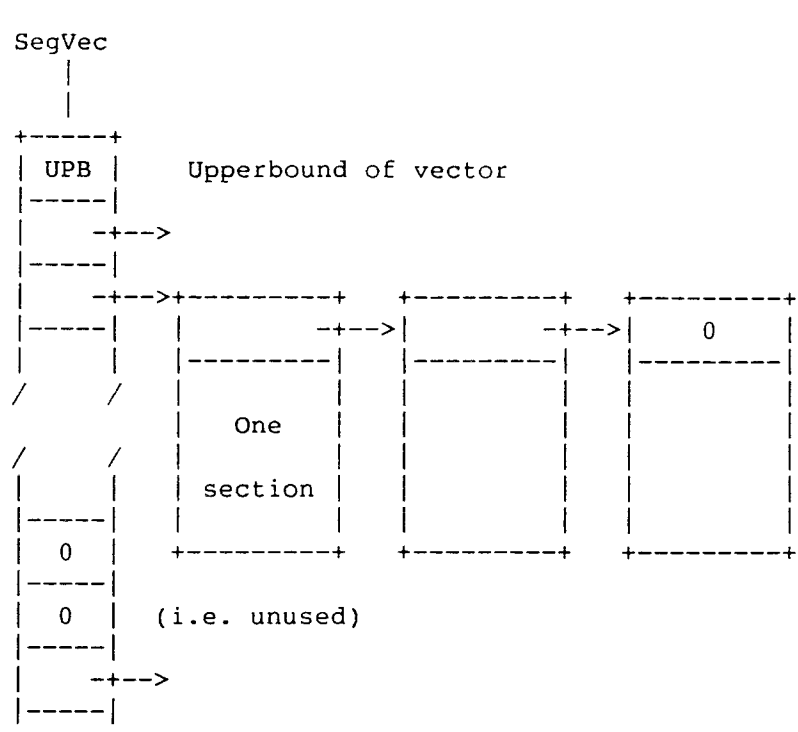
Flags

These also indicate states of the task, but do not affect scheduling, and so live in a separate word. They are set and tested by using the kernel primitives SetFlags and TestFlags. They are useful as a cheap signal between tasks, and are used to implement break.

The console handler responds to CTRL-C, CTRL-D, CTRL-E, and CTRL-F by setting flags #B0001 to #B1000 respectively in the currently selected task. Conventionally, the #B0001 flag is recognized by commands and causes them to finish. Flag #B0010 is inspected by the CLI between commands, and causes termination of command sequences.

SegVec

This pointer leads to all the program sections which comprise the code of the task. The TCB pointer addresses a vector, each element of which either points to a chain of sections, or is zero.



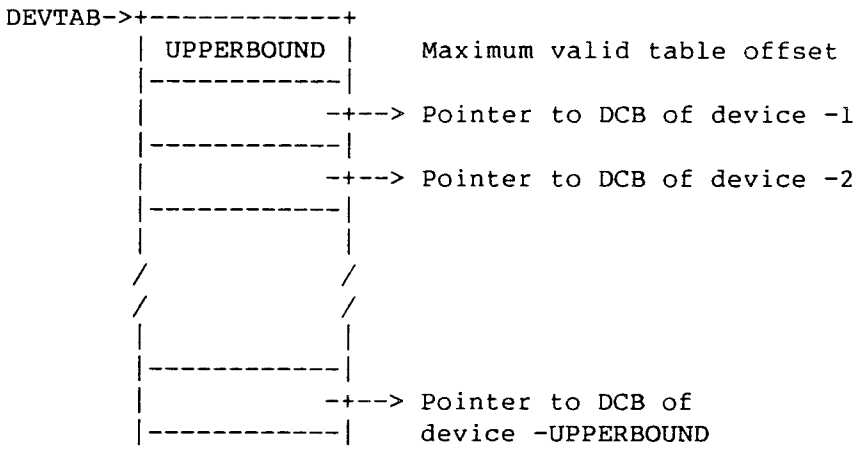
Conventionally, the first offset is used for the Tripos system library, and the second is used for code specific to this task.

3.2.8 TCBList

As well as being addressed by the task table, the TCBs are linked into a chain, for the benefit of the scheduler. The TCBList field of the rootnode points to the TCB of highest priority, whose link field points to the TCB of next highest priority. The chain links all the TCBs in order of decreasing priority, ending with that of the idle task, which has a priority of zero, and link of zero. No two tasks may have the same priority, so the correct chain order is well defined.

The scheduling rule is very simple: the highest priority task which is free to run is the one that should be running. A task is free to run if it is not held, and, if it is waiting or dead, its work queue is not empty. Whenever the task selector is entered, it is handed the TCB of the highest priority task which might be runnable. If it cannot run this task, it just chains down the TCB list until it finds one that it can run. The idle task is always free to run, so the task selector cannot fall off the end of the chain.

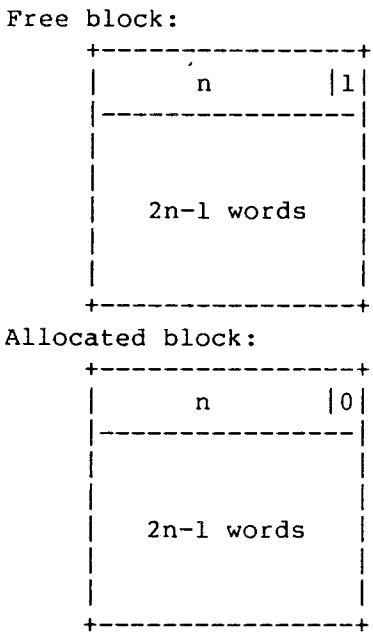
3.2.9 Device Table



This table is used to find the device control block from the corresponding device id. Unused locations contain zero. Device ids are negative integers; the value -1 is used for the clock.

3.2.10 Free Memory Allocation

Blocks of memory are allocated and freed by the kernel primitives GetMem and FreeMem. Memory is allocated from an area which starts at the address given by the BLKLIST field of the rootnode. This area is divided into contiguous blocks, which each consist of an even number of words. The first word of each block is used both to indicate the length of the block (and hence the start of the next), and to record whether or not the block is allocated. As all block lengths are even, the least significant bit of the length is not needed, and so this is used to indicate allocation:



The end of the block list is marked by a word containing zero.

3.2.11 Info Substructure

To access a further substructure with the following format, you use the Info pointer.

Value	Function	Description
BPTR	McName	Network name of this machine
BPTR	DevInfo	Device list
BPTR	Segments	Resident Segment List
BPTR	Devices	Resident Device List
APTR	NetHand	Unused; currently zero

DevInfo

The DevInfo structure is a linked list. You use it to identify all the device names that Tripos knows about; this includes ASSIGNed names and disk volume names. There are two possible formats for the list entries depending on whether the entry refers to a disk volume or not. For an entry describing a device or a directory (via ASSIGN) the entry is as follows:

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
LONG	Type	List entry type (device or dir)
APTR	Task	Handler task or zero
BPTR	Lock	File system lock or zero
BSTR	Handler	File name of handler or zero
LONG	StackSize	Stack size for handler task
LONG	Priority	Priority for handler task
LONG	Startup	Startup value, pass to handler
BPTR	SegList	SegList for handler task or zero
BPTR	GlobVec	Unused
BSTR	Name	Name of device or ASSIGNed name

The Next field links all the list entries together, and the name of the logical device name is held in the Name field.

The Type field is 0 (dt_device) or 1 (dt_dir). You can make a directory entry with the ASSIGN command. This command allocates a name to a directory that you can then use as a device name. If the list entry refers to a directory, then the Task refers to the file system task handling that disk, and the Lock field contains a pointer to a lock on that directory.

If the list entry refers to a device, then the device may or may not be resident. If it is resident, the Task identifies the handler task, and the Lock is normally zero. If the device is not resident, then the Task is zero and Tripos uses the rest of the list structure in the following way.

If the SegList is zero, then the code for the device is not in memory. The Handler field is a string specifying the file containing the code (for example, SYS:L/RAM-HANDLER). A call to LoadSeg loads the code from the file and inserts the result into the SegList field.

Tripos now creates a new handler task with the SegList, StackSize, and Pri values.

The new task is passed a message containing the name originally specified, the value stored in Startup and the base of the list entry. The new handler task may then decide to patch into the Task slot the task id or not as required. If the task slot is patched, then subsequent references to the device name use the same handler task; this is what the RAM: device does. If the task slot is not patched, then further references to the device result in new task invocations.

The DEV.STARTUP field in the DevInfo structure has the following structure for a device running the file system (for example, DFn and anything placed there by the MOUNT command):

Value	Function	Description
LONG	Unit	Unit number
BSTR	DevName	BSTR to device name
BPTR	Envec	BPTR to ENVEC

An ENVEC has the following structure:

Value	Function	Description
LONG	Size	Size (always 11)
LONG	BSize	Size of block (always 128)
LONG	SecOr	Sector origin (not used, always 0)
LONG	Surfno	Number of surfaces (=> 1)
LONG	Secno	Not used (always 1)
LONG	Blkno	Number of blocks per track
LONG	RBlkno	Number of reserved blocks (2 for bootstrap disks)
LONG	Palloc	Preallocation factor (not used, 0)
LONG	Interl	Interleave factor (normally 0, only used when writing)
LONG	Lcn	Lowest cylinder number
LONG	Hcn	Highest cylinder number
LONG	Cblkno	Number of cache blocks needed (=> 3)

If the type field within the list entry is equal to 2 (dt_volume), then the format of the list structure is slightly different.

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
LONG	Type	List entry type (volume)
APTR	Task	Handler task or zero
BPTR	Lock	File system lock
LONG	VolDays	Volume creation date
LONG	VolMins	
LONG	VolTicks	
BPTR	LockList	List of active locks for volume
LONG	DiskType	Type of disk
LONG	Spare	Not used
BSTR	Name	Volume name

In this case, the name field is the name of the volume, and the Task field refers to the handler task if the volume is currently inserted; or to zero if the volume is not inserted. To distinguish disks with the same name, Tripos timestamps the volume on creation and then saves the timestamp in the list structure. Tripos can therefore compare the timestamps of

different volumes whenever necessary.

If a volume is not currently inserted, then Tripos saves the list of currently active locks in the LockList field. It uses the DiskType field to identify the type of disk, currently this is always a Tripos disk. The disk type is up to four characters packed into a longword and padded on the right with nulls.

Resident Segment List

This is a list of segments held in memory either as part of the resident Tripos image, or because they have been brought into memory and added to this list by a CLI command. Each list entry includes a segment name, a use count and a pointer to the segment list.

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
BSTR	name	Name of this segment
LONG	Use	Use count
BPTR	SegList	Pointer to segment list

Resident Device List

This list is very similar to the segment list described above, but in this case it contains all the devices currently available within the system, and which have been installed via a call to AddDevice. Each list entry holds a device name, a use count and a device identifier.

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
BSTR	name	Name of this device
LONG	Use	Use count
LONG	DeviceID	Identifier for this device

3.3 File Info Structure

The structure of the FileInfoBlock data area manipulated by Examine and ExNext is as follows:

Value	Function	Description
LONG	DiskKey	Key number of the object
LONG	DirEntryType	Primary entry type
BYTE	FileName[108]	Object name in the form used by BSTR
LONG	Protection	Protection flags
LONG	EntryType	Secondary entry type
LONG	Size	File size in bytes
LONG	NumBlocks	Number of filing system blocks occupied
DateStamp	Date	Days,Mins,Ticks of the creation date
BYTE	Comment[116]	Object comment in the form used by BSTR

3.4 Segment Lists

To obtain a segment list, you call LoadSeg. The result is a BPTR to allocated memory. The length of the memory block is 4 more than the size of the segment list entry, allowing for the link field.

The SegList is a list linked together by BPTRs and terminated by zero. The remainder of each segment list entry contains the code loaded. Thus the format is

Value	Function	Description
LONG	NextSeg	BPTR to next segment or zero
LONG	FirstCode	First value from binary file

3.5 File Handles

File handles are created by the Tripos function Open, and you use them as arguments to other functions such as Read and Write. Tripos returns them as a BPTR to the following structure.

Value	Function	Description
LONG	Link	Not used
LONG	Interact	Boolean, TRUE if interactive
LONG	ProcessID	Task id of handler task
BPTR	Buffer	Buffer for internal use
LONG	CharPos	Character position for internal use
LONG	BufEnd	End position for internal use
APTR	ReadFunc	Function called if buffer exhausted
APTR	WriteFunc	Function called if buffer full
APTR	CloseFunc	Function called if handle closed
LONG	Arg1	Argument
LONG	Arg2	Argument

Most of the fields are only used by Tripos internally; normally Read or Write uses the file handle to indicate the handler task and any arguments to be passed. Values should not be altered within the file handle by user programs, except that the first field may be used to link file handles into a singly linked list. The file handle type affects Arg1 and Arg2.

3.6 Locks

The filing system extensively uses a data structure called a lock. This structure serves two purposes. First, it serves as the mechanism to open files for multiple reads or a single write. Note that obtaining a shared read lock on a directory does not stop that directory being updated.

Second, the lock provides a unique identification for a file. The lock contains the actual disk block location of the directory or file header and is thus a shorthand way of specifying a particular file system object.

The structure of a lock is as follows.

Value	Function	Description
BPTR	NextLock	BPTR to next in chain, else zero
LONG	DiskBlock	Block number of directory/filehdr
LONG	AccessType	Shared or exclusive access
APTR	ProcessID	Process ID of handler task
BPTR	VolNode	Volume entry for this lock

Because Tripos uses the NextLock field to chain locks together, you should not alter it. The filing system fills in the DiskBlock field to represent the location on disk of the directory block or the file header block. The AccessType serves to indicate whether this is a shared read lock, when it has the value -2, or an exclusive write lock when it has the value -1. The ProcessID field contains a pointer to the handler task for the device containing the file to which this lock refers. Finally the VolNode field points to the node in the DevInfo structure that identifies the volume to which this lock refers. Volume entries in the DevInfo structure remain there if a disk is inserted or if there are any locks open on that volume.

Note that a lock can also be a zero. The special case of lock zero indicates that the lock refers to the root of the initial filing system.

Chapter 4: Installation

This chapter describes how to install Tripos on a new computer. This ranges from simply installing a new type of VDU to writing a complete set of device drivers for a new piece of hardware.

Table of Contents

4.1 Introduction

The Tripos operating system consists of a number of devices drivers and tasks. Each hardware peripheral has a device associated with it; a device normally consists of a device driver and a device handler. The device driver is a very low-level routine that communicates with the hardware; the device handler is a task that provides a high-level interface between other tasks in the system and the device driver.

This chapter describes how to install Tripos on a new computer. At the very simplest this means installing a particular make of VDU; this process is described in Section 4.2, "VDU Installation." A more complex case is installing a new disk drive on an existing Tripos implementation. The MOUNT command is used to do this, and the format of the specification files used by MOUNT are described in Section 4.3.

A more complex case is when Tripos is being moved to a new make of computer. In this case you must learn how to create a new system image and how to write new device drivers. The rest of this chapter describes how to do this.

Besides a variable number of tasks and devices, a Tripos computer contains system libraries, exception handlers, and a small absolute area of memory known as the rootnode. The layout of the rootnode is described in Section 3.2, "Global Data Structure." The contents of a Tripos system image, which is loaded into memory when a system is booted, is constructed by a program called SYSLINK. This program and the process of system generation is described in Section 4.4.

The design of device drivers is described in Section 4.5; this covers the standard commands that must be supported by a device driver as well as the extra commands required by certain specific devices such as serial lines and disks. A complete example of a serial device and a disk device is included.

Section 4.6 covers the design of a device handler task. In most Tripos systems this is not required, as standard handler tasks exist for disks, serial and parallel lines and primary console. However, if a special

purpose peripheral is to be included, then this section describes the structure that is needed to provide a new device handler.

Section 4.7 provides a brief outline of how to port Tripos to your machine.

4.2 VDU Installation

Tripos offers a number of commands that use specific actions from a reasonably intelligent VDU (for example, ED). Unfortunately the control codes for these actions can differ from terminal to terminal.

In order to provide support for a number of different terminals, Tripos allows the user to specify which type of terminal they are going to use. The command VDU <name> sets up the terminal as <name> type.

The VDU command works by reading a file called DEVS:VDU and constructing, from the specification found there, a section of interpreted code. This code is stored in the console task associated with the CLI. Look at the file (using TYPE) and determine if your terminal is already in there. If so, you are in luck. All that you need to do is to issue the command VDU followed by the type of terminal you are using. For example,

```
VDU tvi
```

if your terminal is a Televideo 950. The VDU command is normally placed in the file SYS:S/STARTUP-SEQUENCE in order to save issuing it every time you startup Tripos. This file contains the commands that Tripos executes each time it is started.

If your terminal is not in the list then you must specify the commands needed to drive it. This involves adding information to the file DEVS:VDU and should not be attempted until you are used to the editor EDIT. You can, of course, use ED if you have another terminal available whose type is already known.

You can see the general layout of the file from those entries already in it. The entry starts with the word 'VDU' followed by the terminal name (maximum 15 characters) followed by a semicolon. There are then two

sections. The first defines the output from programs such as ED, while the second describes any changes you may wish to make to characters entered as input to ED. Each section starts with the word 'output' or 'input' and is terminated by 'end', followed by a semicolon.

The output section contains a number of action names which may be entered in any order. Each name may be followed by a number of directives which tell the system what to do. The most common is the word 'send', followed by a list of values separated by commas. Each value is sent to the terminal when the editor wishes to execute the required action. Values can be specified as a number in decimal, a number in hexadecimal preceded by 'H', a control character if a character is preceded by '^' (up arrow), or as an actual character enclosed in single quotes ('). Thus, in order to move the cursor down, your terminal might need to be sent linefeed (Hex 0A or CTRL-J). You would specify this as one of the following:

```
cursordown send H0A;
cursordown send ^J;
cursordown send 10;
```

Alternatively, your terminal might need to be sent Escape followed by the letter 'D' as follows:

```
cursordown send H1B,'D';
```

Each name and associated set of directives must be terminated by a semicolon. There are three other possible directives. The first two directives represent the X (or column) and Y (or row) position of the cursor; these are 'xpos' and 'ypos'. ED assumes that position 0,0 is the top left hand corner of the screen; many terminals require a fixed offset to be added to the value sent to position the cursor. This value must be given even if it is zero. Thus you might position the cursor by sending CTRL-P followed by the X position offset by 32, then the Y position offset by 32. For example:

```
setcursor send ^P xpos 32 ypos 32;
```

Alternatively, you might have to send escape, an equals character, then the Y position followed by the X position with no offset. For example:

```
setcursor send H1B,'=' ypos 0 xpos 0;
```

Finally, some action requests may involve moving the cursor as a side effect; for example, scrolling the screen up by one line. In this case, you should not send the actual control codes to move the cursor as ED thinks it knows where the cursor is on the screen. Instead, you should use the directive 'goto' followed by the X position, then the Y position. This tells ED to position the cursor and to keep track of where it is on the screen. Eventually, of course, ED uses the values defined to physically move the cursor. Thus, to scroll up the screen, you might specify the following: move to the bottom line and execute a linefeed. In other words,

```
scrolldown goto 0,23 send H0A;
```

The action requests are as follows.

init
This is called before any other calls and allows the terminal to be set up (for example programming function keys). It is optional.

uninit
This is called when ED has finished and allows you to undo anything done in the init section, such as switching back to roll mode from page mode. It is optional.

length, width
These two specify length and width of the terminal and if omitted default to 24 lines and 80 characters respectively. If used they must be followed by the directive 'return', then the value required, then the semicolon.

highlighton, highlightoff
The 'highlighton' action is used to make the terminal highlight any text sent to the screen until a 'highlightoff' action. The effect on the screen may be chosen to be any special effect such as inverse video, underlining or whatever. The only proviso is that the effect must not take up a character position. The action can be omitted if your terminal cannot do this.

insertchar, deletechar
Some terminals are capable of making room for a character in the middle of a line by moving any characters to the right up by one place. They can usually also delete a character by moving the characters to the right back towards the start of the line to close up the line of text. If your terminal supports this, then the values to be sent should be entered here; otherwise the action names can be omitted and ED rewrites the entire line where required.

insertline, deleteline, deol, clearscreen
These are the actions of inserting a line and shuffling text down, deleting a line and shuffling text up, deleting all characters from the cursor position to the end of the line and clearing the entire screen. These actions must be defined. Often inserting and deleting lines can be done on terminals with split pages by splitting the page and scrolling one of the halves to get the correct effect.

cursorup, cursordown, cursorleft, cursorright
These are the action requests to move the cursor one place at a time. They must be defined.

setcursor
This action routine is called to move the cursor by more than one place. The specification generally uses the 'xpos' and 'ypos' directives.

scrollup, scrolldown
These action routines enable ED to scroll the text on the screen up or down. Some terminals take much longer doing it one way (such as deleting a line at the top of the screen for scrollup) than another (such as performing a reverse line feed at the top of the screen). Scroll down can often be implemented by sending a linefeed at the bottom of the screen. These routines will often contain the 'goto' directive and they must be defined.

The input routine allows multiple control codes sent by a terminal to be mapped to a single control code to control the editor. For example, a cursor right key might send CTRL-L when ED expects CTRL-X to be the code to do this. To get around this problem, you can include the line

```
map ^L to ^X;
```

in the input section. The line must start with the word 'map', which is followed by a number of input characters transmitted from the terminal. The single value which is to be sent to ED is given after the word 'to'. Multiple input sequences are allowed, when, for example, a function key sends escape followed by one or more characters. In this case the line might read

```
map H1B,'A' to ^J;
```

Notice that if you map out a control code normally used by ED, then you must provide a suitable replacement. The above example makes Escape by itself unavailable, so you should define some character sequence to map to Escape. Character sequences starting with the same control code must be of the same length to avoid ambiguity. Only control codes may be mapped. Do not attempt to map the control codes CTRL-S and CTRL-Q onto anything as these are defined to be flow control characters which stop and start output from Tripos.

A complete example follows.

```
Vdu Newbury;
  Output;
    length return 24;
    width return 80;
    scrollup goto 0,23 send H0A;
    scrolldown goto 0,0 send H01;
    setcursor send H16 xpos H20 ypos H20;
    insertline send H01;
    deleteline send H02;
    highlighton send H12;
    highlightoff send H13;
    insertchar send H0F;
    deletechar send H0E;
    deol send H19;
    cursorup send H0B;
    cursordown send H0A;
    cursorleft send H08;
    cursorright send H0C;
    clearscreen send H1F;
  End;
  Input;
    map H0C to ^X;
  End;
```

In addition, you may wish to set the serial line parameters. This is done by the SET-SERIAL command. (See Chapter 1, "Tripos Commands," of the *Tripos User's Reference Manual* for a specification of this command.) If you always wish a particular output port to have certain parameters, then a suitable SET-SERIAL command can be placed into the file called SYS:S/STARTUP-SEQUENCE. This file contains commands which Tripos executes each time it is started; it commonly contains commands to configure the terminal such as CONSOLE as well as SET-SERIAL commands for the serial lines.

MOUNT

When Tripos is booted the system knows about a number of devices which correspond to physical peripherals. A system might start with DF0: as a floppy disk, DH0: as a hard disk, SER: as the serial port and so on. A list of available devices can be obtained from the ASSIGN command.

Further devices can be made available with the MOUNT command. This command reads a specification file and makes the device you specified available for use. The device is not actually initialized until it is first referenced. (A standard device, such as SER:, is automatically MOUNTed, and is only initialized when first referenced.) A brief specification of MOUNT can be found in Chapter 1, "Tripos Commands," in the *Tripos User's Reference Manual*.

The file which MOUNT reads giving details of available devices is called DEVS:MOUNTFILE. The information presented in the file is placed into a new entry in the DevInfo structure described in the previous chapter. This file has entries of the form:

<devicename> <arguments> #

separated by spaces or newlines. A device name is a name followed by colon (for example, HD0:) and the arguments consist of keywords followed by equals followed by a value. For example:

```
HD0: Device = disk0
```

Each argument may be on a separate line, or on the same line if terminated by a semicolon.

Comments may be added to DEVS:MOUNTFILE if they are enclosed in /* */ pairs. For example:

```
Reserved = 2 /* Not to be used for storage */
```

The argument keywords and their expected values are as follows:

Keyword	Description
Device	The name of the disk driver device
Unit	The unit number to be used
BytesPerBlock	The size of each block in bytes
SectorOrigin	The lowest sector number on a track
SectorsPerBlock	Number of sectors in a block
Interleave	The interleave factor
LowCyl	The lowest cylinder number
HighCyl	The highest cylinder number
Surfaces	The number of surfaces on the disk
BlocksPerTrack	The number of blocks on each track
Reserved	The number of reserved blocks (not for storage)
Buffers	The number of cache buffers to be used

Notes:

1. A large disk may be partitioned by using different values for LowCyl and HighCyl on the same disk.

2. More than one description can be placed on a line if the separator character ';' is used.

3. Numbers are assumed to be decimal unless they are preceeded by 0x, in which case they are treated as hex.

4. Comments may be placed in the file in the same way as in C.

5. Descriptions are separated by the # character.

6. The '=' character associating a value with a keyword is optional.

7. The SectorOrigin value is often 0 or 1. Commonly one sector is used for one block, so that SectorsPerBlock is 1. The Interleave factor is only used when writing data onto disks, and defaults to 1 if omitted. An interleave factor of n will cause consecutive blocks of a file to be written into sectors s, s + n and so on.

For example:

```
DF0:        Device          = floppydisk
            Unit            = 0
            Surfaces        = 2
            SectorOrigin    = 0
            BytesPerBlock   = 1024
            SectorsPerBlock = 1
            BlocksPerTrack  = 4
            Reserved        = 1       /* For bootstrap */
            Interleave      = 0
            LowCyl = 0  ;  HighCyl = 79
            Buffers = 5
 #
```

The algorithm that the file handler uses to convert a byte offset to the disk into the track, sector surface, and so on is as follows. Imagine the disk to be made up of several disks, each with an upper and lower side, where each disk has a series of tracks, starting at the outer edge of the disk and continuing towards the centre, and each track contains a number of blocks (0-n). The order starts with Side 0 (the upper surface of the top 'disk'), Block 0 (the first block), Track 0 (the outermost track), continues through Blocks 0-n on Track 0, Side 0, then through Blocks 0-n on Track 0, Side 1 and so on down to Side n. This is followed by Blocks 0-n, Track 1, Sides 0-n, Blocks 0-n, Track 2, Sides 0-n, and so on to Track n.

4.4 System Generation

A Tripos operating system image consists of a number of executable binary segments, which may have been produced by the assembler or by a compiler. These segments are copied by the system linker SYSLINK into a Tripos operating system image file that can then be installed with the INSTALL command. The SYSLINK program builds the system image file according to a set of instructions. For instance, the program is instructed as to the files that should be included, the size and layout of the memory of the target machine, and the details of the filing system that is to be used.

The command template for SYSLINK is as follows:

```
SYSLINK "FROM/A,TO/A,MAP/K,OPT/K"
```

The FROM argument should be the name of a file of commands for the linker describing the system to be built, and the TO argument should be the name of the required system image file. The MAP keyword may be used to obtain a map of the system; in which case it should be followed with a suitable filename for the mapping output. The OPT keyword may be used to supply options; the letter 'W' followed by an integer specifies the workspace size, and 'F' specifies full map output, rather than the abbreviated form given by default. The 'S' option tells SYSLINK to produce an output file in Motorola S-Record format rather than the standard Tripos binary format.

The action of SYSLINK is to read the instructions given to it, and to produce a complete system image file. This file is specific to the hardware of the target machine, and is in the form of a large section of code containing a number of absolute references. Any relocatable sections are relocated for the address in memory allocated in the target machine by SYSLINK.

All code sections are allocated in such a way that they resemble segments loaded by the library routine LoadSeg, and thus UnLoadSeg may be used on them later if required. This is useful in reclaiming the space used by initialization segments, for example. Control blocks and other system structures such as the rootnode are preallocated and initialized.

SYSLINK also handles the layout of absolute store, and ensures that sections of assembler specified as absolute are loaded in the correct place. This enables interrupt locations and TRAP vectors to be set up when the system image file is loaded. All unallocated store is set up so that GetMem and FreeMem operates correctly without any further initialization.

The linker operates in three phases: first, it reads the command file, checking for syntax errors and building a tree structure representing the declarations; second, it scans the tree checking the validity and consistency of the declarations; third, it reads the required code segment files and writes out the system image file.

The syntax for the command file is as follows:

```
<commandfile>  ::= <declaration>; [<declaration>;]*
<declaration>  ::= <star><declaration>
               ::= <segment-decl>
               ::= TASKTAB <n>
               ::= <task-decl>
               ::= DEVTAB <n>
               ::= <device-decl>
               ::= INFO <info-decl>
               ::= ABSMIN <n>
               ::= ABSMAX <n>
               ::= TCBSIZE <n>
               ::= STOREMIN <n>
               ::= STOREMAX <n>
               ::= MCADDRINC <n>
               ::= ROOTNODE <n>
               ::= MEMORYSIZE <n>
<segment-decl> ::= SEGMENT <name> <name-list>
<task-decl>    ::= TASK <n> PRIORITY <n>
                   STACK <n> SEGMENTS <name-list>
<device-decl>  ::= DEVICE <n> DRIVER <name>
<name-list>    ::= <name> [, <name>]*
<info-decl>    ::= ( <info-decl> | <info-list> )
<info-list>    ::= [<info-elem> [, <info-elem>]*]
<info-elem>    ::= <name> | <n> | <string>
```

Layout characters space, tab, and newline are ignored, except that they terminate names and numbers. <star> is the character '*'. <n> is a string of one or more digits that may be preceeded by '#' to denote an octal number, or by '#X' to denote a hexadecimal number. <name> is a string of one or more characters, except layout characters, ';', ',', and not starting with a digit. SEG is allowed as an abbreviation for SEGMENT, SEGS for SEGMENTS, PRI for PRIORITY, and DEV for DEVICE. Keywords are accepted in either case.

4.4.1 Memory Specification

The initial part of a command file specifies the size and layout of memory. The declarations ABSMIN and ABSMAX define the limits of absolute store, parts of which may be defined by absolute sections of assembler code read during the link process. STOREMIN and STOREMAX define the limits of available memory that may be used as free store. The memory thus defined are allocated by SYSLINK from both high and low memory address. Low store is used for system data structures, while high store is used for code segments. The area defined by ABSMIN and ABSMAX may not overlap with that defined by STOREMIN and STOREMAX. The value MEMORYSIZE is used to specify the complete amount of memory available on the target computer - in particular this value is used to determine valid memory references in DEBUG. The number after MEMORYSIZE should be size of the memory in units of 1K BCPL words - thus a 256K machine would be specified as 64.

When Tripos starts up the initialization section of the kernel starts at the value specified by MEMSIZE and checks to see if there is any memory beyond that specified. If so then the free list and the value of MEMSIZE are modified to use all the available store. This means that it is normal to inform SYSLINK that there are only 100Kbytes of memory available, and to use a further 28K for the system bootstrap. Thus Tripos may be booted into a machine with 128Kbytes of memory or more, and will determine how much memory is actually present when it starts.

The location of the rootnode, which is the central Tripos data structure, is specified after the keyword of the same name as a BCPL address. This

must match the value returned by RootStruct() which is specified in the Tripos kernel.

The size of each Task Control Block (TCB) is given in BCPL words after the keyword TCBSIZE. Again this must match the value specified in the kernel.

Finally the keyword MCADDRINC is used to specify the factor by which BCPL addresses must be multiplied to obtain a machine address. For the 68000 this is always 4.

4.4.2 Task and Segment Declarations

A segment declaration declares a segment name which will be used later in the SYSLINK file. It is followed by a number of filenames, separated by commas. Each segment name represents one or more binary files provided as part of the Tripos kit. The system library is specified here, and is made available to every task in the system. A segment is only included in the system image file if it is used as part of a task specification, and it is only ever included once. Multiple references to the same segment share the code. However, identical files included as part of the definition of different segments may be included more than once, and these files are not shared.

If the SEGMENT description is preceded by an asterisk ('*'), then the segment is taken as an initialization segment. This simply means that the segment is placed lower down in memory than resident segments, all segments being allocated store from high to low addresses. Thus if the initialization segment is then UNLOADSEGed after its job is complete, the space returned is coagulated with the rest of free store, rather than leaving a 'hole' in memory that is otherwise permanently allocated.

A TASKTAB declaration sets the size of the task table; otherwise a default of 10 is assumed. This value limits the maximum number of tasks available in a Tripos system.

Each task that is to be part of the initial system must be specified with the TASK declaration. This declaration must be followed by the task number, the priority follows the keyword PRIORITY and the stack size is

given in longwords after STACK. Note that this stack size is the size given to the root stack of the task. Any coroutines run from the root stack have their own stack size specified at that time. In particular, all commands run from the CLI are run as a coroutine, and the stack size for this is specified by means of the STACK command.

The final part of a task declaration is the segment list. This refers to code segments defined via a SEGMENT directive. The keyword SEGMENTS is followed by a list of segment names, with the system library segment specified first, and the segment specific to the task specified second.

Finally, if the task declaration is preceded by an asterisk ('*'), then the task in question is identified as the initial task that is started when the Tripos image is entered. This asterisk should normally precede the declaration of task 1.

It would be possible to include any other tasks required in the system here, but most other tasks which may be needed are created dynamically once the system is running. If an extra task is added here then it would need to be woken up by some other task sending a suitable packet to it.

4.4.3 Device Declarations

The next stage in describing a Tripos image is to specify the devices that are to be used. Again, devices can be created dynamically with the AddDevice primitive, but it is normal for at least a disk device and a terminal device to be declared as part of the initial image.

The size of the device table limits the maximum number of devices available in Tripos in the same way as the size of the task table limits the number of tasks. This size is specified by the DEVTAB declaration. If it is omitted is defaults to 10.

Each device is declared by means of the DEVICE keyword. This is followed by the device number. The devices are normally identified as negative numbers in Tripos, with task numbers being positive. This allows task 0 to represent the idle task. By convention, device -1 is always the clock device. The device number is followed by the keyword DRIVER and the name of the file in which the driver code is to be found.

This code file is another binary load file such as may be produced by the linker ALINK.

4.4.4 The INFO Substructure

The rootnode is the basis of all Tripos data structures, and contains a number of fixed locations which are used within Tripos. However, one of the slots in the rootnode is a pointer to a further substructure that represents other, variable data structures. This structure is described in full in the previous chapter, but parts of it may be preallocated by SYSLINK. The INFO keyword is followed by a list enclosed in brackets. Each member of the list may be a further set of brackets or a number, string, or name. Each bracket represents a further substructure that is pointed at by the list element preceding it.

This is the most complicated part of the SYSLINK file, and is best described with reference to the example in the next section. An overview of the structures that must be defined is given here. The first entry is the machine name - this can be any name enclosed in quotes and is currently unused. It will be used in network systems. The next entry is the DevInfo structure. This contains the names of all the devices which will appear in the system when booted. Other file system devices are made available through the MOUNT command. The devices must include CON: as this is the device which the startup code attempts to open. It must also have a disk device specified. More than one disk device may be specified, but a disk device must be the first element in the list, which means the last entry in the section in the file. This disk device is used as the system disk, and by altering the disk device here you can alter the disk which is used by default. For example, two versions of system image could be produced; one would use a floppy disk as the system disk and one would use the hard disk.

The entry for the CON: device should be copied from the example directly, and any other serial lines to be supported should also be entered with different unit numbers or device names. The device name specified here is a cross reference to the devices named in the device list.

Any disk entry must contain the file system startup information. This is described in Chapter 3, and is a pointer to a table of disk parameters specifying such information as the number of tracks, sectors, surfaces and so on. The information is represented in the SYSLINK file as another bracketed entry containing the parameters. Thus in order to change the size of the system disk the disk parameters would have to be changed.

The next entry in the Info structure is the resident segment list. This must contain entries for the syslib, cli and restart sections. It can also contain any other segments which you would like resident, along with an initial use count.

In a similar fashion the next entry, the resident device list, contains the names and device numbers of resident devices. The entry for the timer must appear, and refer to device -1. This matches up with the actual code specified in the device section where the code for device -1 is handled. Similarly the driver named as that being used by CON: must appear here, as must the name of the disk driver referenced as the device driver used for the disk device.

4.4.5 Example

In the following example, a normal Tripos image is produced. There are two tasks specified; task 1 being the initial CLI and task two the system debugger.

In this example, bold typeface is used for the SYSLINK directives, upper case for file names and lower case for SYSLINK defined names such as those for segments.

```
| Tripos link file

absmin          #X0000;
absmax          #X032F;
storemin        #X4000;
storemax        #X8FFF;
memorysize        36;
rootnode         512;
tcbsize           29;
mcaddrinc          4;
```

The first part of the file specifies various constants, most of which should not be changed. This example places the absolute memory, including the rootnode and exception vectors, from 0 to 32F longwords. The main memory used as heap starts at 4000 longwords ($10000 byte address) and extends up to 8FFF longwords. Tripos will attempt to resize itself into any memory which can be found beyond the end of this high limit. The value of memorysize is set to 36, being the number of 4Kbyte blocks in the heap memory.

In this example the rootnode is placed at BCPL address 512 ($800 byte address), the size of each TCB is 100 longwords and the address increment is 4.

```
seg syslib      klib.obj,
                jacket.obj,
                mlib.obj,
                dlib.obj,
                blib.obj,
                extras.obj,
                io.obj;
```

The segment lists come next. The first segment list specifies the system library syslib. The contents should be kept as supplied, and not changed.

```
seg debug       taskint.obj,
                debug.obj,
                debug-disasm.obj;
```

The debug segment is specified as containing a task interface module and the the two debug code modules. A smaller, cutdown debugger can replace the full version.

```
seg cohand      taskint.obj,cohand.obj;
seg cli         taskint.obj,cli.obj,cli-init.obj;
seg fihand      taskint.obj,
                access.obj,
                bitmap.obj,
                bufalloc.obj,
                disc.obj,
                exinfo.obj,
                main.obj,
                support.obj,
                work.obj,
                moveb.obj,
                state.obj,
                init.obj;
 seg restart    taskint.obj,
                restart.obj;
```

The console handler, cli, file system and file system restart segments are specified in a similar fashion. These segment declarations should not normally be altered.

```
 tasktab 20;

*task   1 pri 1000 stack 400 segs syslib,cli;
 task   2 pri 2000 stack 300 segs syslib,debug;
```

The task declaration section defines task 1 as the cli task, using the system library segment and the cli segment. Task two is defined as the debugger is a similar fashion.

```
 devtab 20;

dev -1 driver bin.clock-driver;
dev -2 driver bin.tty-driver;
dev -3 driver bin.disc-driver;
```

The device declaration section is where you must start to specify the devices which you have written. In this example there are only three devices, being the minimal sytem. These devices are the clock device, the serial line (tty) device and a generic disk device for both floppies and hard disks. Remember that the clock device must be device -1.

```
info ( | Machine name
       "68000",
       | Initial device configuration
       ((((0,
             0, 0, 0, 0, 300, 3000,
               (0, "serial"), cohand, 0, "CON"),
             0, 0, 0, 0, 300, 2999,
               (1, "serial"), cohand, 0, "AUX"),
             0, 0, 0, 0, 210, 2500,
               (0, "disc", (11,256,0,2,1,4,4,0,
                            0,0,79,5)),
               fihand, 0, "DF0"),
             0, 0, 0, 0, 210, 2499,
               (1, "disc", (11,256,0,4,1,8,1,0,
                            0,0,359,5)),
               fihand, 0, "DH0"),
       | Segment list
       (((0, "restart", 1, restart),
             "cli",     1, cli),
             "syslib",  1, syslib),
       | Device list
       (((0, "timer",   1, -1),
             "serial",  1, -2),
             "disc",    2, -3),
       | Spare
       0
     );
```

The final part of the file describes the Info structure which is required. Each entry nested in brackets creates a pointer to the entry in memory. Thus there are five main entries in this example, where many entries contain further pointers. The first main entry is the name of the machine. Although useful for a network machine it is not otherwise used. It can be any string enclosed in quotes.

The second entry specifies the DevInfo substructure which describes the devices that are available in the system. This is specified as a list, so that the four starting brackets represent the four elements in the list. The last list element is the first described, and is the definition for the CON: device. This takes a stack of 300 long words and a priority of 3000. The startup information is a pointer to two longs, the first being the unit number (0) and the second being the name of the device ("serial"). This name is a reference to the name given to the tty device -2 in the Device list later on. The next entry in the CON: description is the segment list name described earlier, and this is followed by an unused slot and finally the name of the device itself.

In a similar fashion the other three entries in the list refer to AUX: (an extra serial line) and two disks, DF0: and DH0: being the floppy and hard disk unit respectively. The system will boot and make DH0: the initial system disk SYS: because it is given as the element at the head of the list (the last to be described).

The entries for the disks are slightly different to that for CON:. The stack sizes required are less and the code segment used is the file system segment. In this implementation both the floppy and the hard disk use the same driver, called "disc" and another reference to the name in the device list. In other implementations the two device drivers would be different. The main difference in the entries for the disks is the startup information, which is the set of parameters required for the disk handler. This is more fully described in the previous chapter, but it includes the size of the disk, number of heads, tracks and so on.

The next main entry in the Info structure is the resident segment list. The syslib, debug and cli segments have already been referenced in the task section, and the cohand and fihand segments have been referenced in the DevInfo substructure above. The file system task requires the restart segment to be held in the resident segment list, while commands such as NEWCLI expect the cli and system library segments to be resident as well. This entry places them there, gives them suitable names and sets the use count to 1.

In a similar fashion the resident device list gives a name to the clock, serial and disk devices. The names are used in the DevInfo structure entries and the device numbers match up with the negative device numbers used in the device specification section.

The final entry is zero, and is reserved for the network device handler. Any more slots required by a particular installer may be specified beyond the current end of this structure.

4.5 Device Drivers

Device drivers connect Tripos to a peripheral. For each peripheral that has an interrupt, there is a device driver: a serial line device driver, a disk device driver, a parallel port device driver, and so on. (All interrupts are handled by drivers.) A device driver consists of two parts: the Device Control Block (DCB), and the actual driver code. This section outlines the DCB, describes the actual driver code, and provides, as an example, a full description of a serial line driver.

4.5.1 Device Control Block (DCB)

The DCB has a structured layout as shown in Figure 4-A below.

```
 0   Head     Pointer to first packet on work queue
 4   Tail     Pointer to last packet on work queue
 8   QAct     Action routine used by QPKT
12   DQAct    Action routine used by DQPKT
16   Pri      Priority of this device
20   Id       Device id of this device
24   Open     Device Open routine
28   Close    Device Close routine
32   StartIO  Device StartIO routine
36   AbortIO  Device AbortIO routine
40   RecallIO Device RecallIO routine
44   NITB     Number of Interrupt Transfer Blocks
```

Each DCB contains all of these locations, and is followed by NITB (that is, the number of Interrupt Transfer Blocks). There is always at least one

Interrupt Transfer block (ITB), and there is an ITB for each different interrupt vector handled by this device. The format of an ITB is as follows.

```
 0   (private)
 6   Offset   Offset of base of DCB
10   Next     Pointer to next ITB using this vector
14   Code     Interrupt routine
18   Vector   Interrupt vector address
22   RecallQ  Recall queue
26   UserData User data area
30   Packet   Packet to be returned
```

The user is responsible for setting up the following fields within a DCB.

- Priority of device
- Number of ITBs
- Each ITB
- Offset and interrupt vector within each ITB
- Routine entry points

4.5.2 Device Driver Code

The entry points within the DCB (see Figure 4-A) are pointers into entry points into the code of the device driver itself. The following entry points must be provided:

- Open
- Close
- StartIO
- AbortIO
- Interrupts
- RecallIO

Each of these is described below.

Open

```
Open( dcb )
```

The Open entry point is the code that gets called when you initialize a device. This is the code that sets up initial data structures, resets the peripheral and whatever else needs to be done upon initialization.

Close

```
Close( dcb )
```

In the same way, Close undoes all the things done by Open.

StartIO

```
StartIO( dcb pkt )
```

The StartIO call is the entry point that is called when a packet arrives for the device. The arriving packet will have a packet type associated with it; an instruction indicating the type of I/O that is required (for example, reading or writing). Normally, upon being received, a packet has to translate the command it contains to hardware-specific commands and then return having set up hardware that causes an interrupt later. When the interrupt does occur, the interrupt handler for the device is called.

The called packet is queued to the head of the driver's work queue and exits from RecallIO if anything is on the driver's work queue.

Note that the function of StartIO is hardware specific.

AbortIO

```
AbortIO( dcb, pkt )
```

The AbortIO entry point is used to cancel a request. There is a standard call inside the kernel called DQPkt and this is used to cancel an I/O request. The action of DQPkt is as follows:

1. It looks at the specified device driver to see if the packet is still on the work queue for the device driver. If it is, it takes it off the work queue and the device driver never gets to see the packet.

2. DQPkt looks at the work queue of the task that is issuing the call to see whether, in fact, the packet has already come back and is sitting on the task work queue. If the packet cannot be found on either of the work queues, then the system calls AbortIO with the address of the packet and this gives the device driver a chance to take the packet of an internal work queue and to cancel any I/O which may be pending.

Interrupt

```
Interrupt( dcb, itb )
```

You should do as little work as possible in the interrupt routine. The code should normally turn off the interrupt and reschedule a call to RecallIO. The Interrupt entry point is called as a hardware interrupt, stopping whatever was going on (possibly a lower priority interrupt) and inhibiting interrupts at this or any lower level. You must not spend too much time in the interrupt handler, nor must you make any kernel calls.

The result from the routine has three possible values. A negative value indicates that this interrupt routine cannot handle the interrupt and the system will then go on to call another interrupt handler in the same chain. A value of zero indicates that the interrupt has been handled but no call to RecallIO is required. This is commonly the case when an interrupt occurs but no packet is waiting. A positive value requests that RecallIO be called. The RecallIO entry point will be called as quickly as possible after the interrupt, but not immediately if the interrupted task was in the middle of a kernel call. The Interrupt routine should ensure that no further interrupts can occur until the RecallIO routine has had a chance to deal with the situation and turn interrupts back on.

Consider transmitting data from a buffer into a serial port. In most cases the Interrupt routine will get called when the hardware is ready to transmit again. In most cases the Interrupt routine will move another character out of the buffer into the hardware, increment a pointer and return zero indicating that no RecallIO is required. When the buffer has

been transmitted the transmit interrupt should be turned off and a call to RecallIO requested. The RecallIO routine will then return the packet which initiated the request. It may also see if any more packets are waiting on some internal queue and handle those if required.

In the same way an interrupt handler for a more complex device such as as a disk would make a request for RecallIO to be called when the disk interrupted after a DMA read or write request had been satisfied.

RecallIO

```
RecallIO( dcb, itb )
```

At the RecallIO entry point, it is normal for the device driver to look at the return codes and send the packet that caused the interrupt back with either a zero or a non-zero return code indicating the result of the request.

Again, consider a disk transfer interrupt, where all that is done in the interrupt routine is to turn the interrupt off. Soon afterwards the RecallIO routine is called, where the status port can be read and the packet involved sent back to the client task.

Once the RecallIO has been called, the kernel sees if there are any packets waiting on the device driver work queue. If so, it calls the StartIO entry point once more so the next request can be handled.

4.5.3 Examples of Device Drivers

The following examples describe the design of three different device drivers. Firstly a disk device driver is presented, with a large amount of commentary describing what is happening. Secondly a more complicated serial line driver is included, and the final example is the special case of the clock device.

Disk Device Driver

The first example is that of a very simple device driver: a disk device driver for a single disk unit, with a very simple hardware interface, using no internal packet queues or units. In this example, you are taken through each step in the design of the driver, and some code is provided to illustrate what you would write.

The first step is to define the Device Control Block or DCB. This begins with a cross reference to all the external names that are needed. For example:

```
        XREF    Open
        XREF    Close
        XREF    StartIO
        XREF    AbortIO
        XREF    RecallIO
        XREF    Int
```

This is followed by the standard DCB structure. The only value that you must specify is the device priority, which indicates the relative priority of this device with respect to others. The structure appears as follows:

```
Base    DC.L    0       Work Queue Head
        DC.L    Base    Work Queue Tail
        DC.L    0       Qpkt Action routine
        DC.L    0       DQpkt Action routine
        DC.L    8       Queue Priority
        DC.L    0       Device identifier
```

Next, you provide a totally standard layout for the entry points. These externals are filled in when Tripos loads the code.

```
        DC.L    Open
        DC.L    Close
        DC.L    StartIO
        DC.L    AbortIO
        DC.L    RecallIO
```

You then place the number of interrupt transfer blocks (ITB), one in this case, followed by the ITB itself. You must remember to fill in the address of the interrupt routine (the external Int) and the memory vector location ($100). Note that this value is the vector offset, not the vector number.

```
DC.L    1         Number of ITB

DS.W    3         (Used by kernel)
DC.W    Base-*    Offset to DCB
DC.L    0         Pointer to next ITB
DC.L    Int       Interrupt routine
DC.L    $100      Interrupt offset
DC.L    0         Recall Q
DC.L    0         User data area
DC.L    0         Packet to be returned
END
```

The second file contains the disk driver. For the purposes of simplicity, the DCB and the driver are in two separate files in this example; however, you can have both of them in one file if you wish. All the functions in the driver conform to the C calling convention, and so although the DCB must be written in assembler the driver may be written in C. The first few lines of the driver should include the standard header file and define the externals referenced in the DCB:

```
INCLUDE "tripos.i"
XDEF    Open
XDEF    Close
XDEF    StartIO
XDEF    AbortIO
XDEF    RecallIO
XDEF    Int
```

After this you provide some standard offsets for the packet structure, the command values, and standard constants:

```
P_UNIT    EQU  P_ARG1
P_BUF     EQU  P_ARG2
P_SIZE    EQU  P_ARG3
P_OFF     EQU  P_ARG4

* Disc driver specific commands

C_FORMAT EQU  4
C_STATUS EQU  5
C_MOTOR  EQU  6

* Standard constants

E_UT      EQU  12      Unknown type error
```

A private data area is kept with information about each unit controlled by this driver. You can define the structure of this area as follows:

```
* Private data area structure

InitD    EQU      0
ActD     EQU      InitD+$1D+1
FormD    EQU      ActD+$15+1
StatD    EQU      FormD+$0F+1
MaxData  EQU      StatD+$07+1

MaxUnit  EQU      2      Maximum number of units supported
```

Next, you define the IO locations that you need. The actual hardware in this example is a Motorola VME Intelligent Disk Controller, MVME315. The details of which are immaterial here.

```
IDC_BASE     EQU  $FF0000
IDC_CMDSENT  EQU  IDC_BASE+$101
IDC_CMDBUF   EQU  IDC_BASE+$105
IDC_MSGSENT  EQU  IDC_BASE+$181
IDC_STATBUF  EQU  IDC_BASE+$185
IDC_STC      EQU  IDC_STATBUF+12
IDC_STATUS   EQU  IDC_STATBUF+16
MCR          EQU  $FE00F1
```

You now come to the first of the entry points, Open. This is called with the DCB as an argument. All system routines may corrupt D0-D3 and A0/A1; these are the registers Lattice C corrupts when a function is called. The job to be done here is to get the device id, which is initialized into the DCB, and copy it into a private area to make it easier to access. Next, you initialize the controller, and then the device. For example:

```
Open    MOVEA.L 4(SP),A1        Get DCB ptr
        MOVE.L  D_Id(A1),Devid  Save my device ID
* Reset the IDC
        LEA.L   InitIDC,A0
        BSR     SendC
* Now Reset all the devices
        LEA.L   Unit0,A1
        BSR     ResetAll
        LEA.L   Unit1,A1
        BSR     ResetAll
        RTS
```

The Close routine, in this case, has nothing to do. It is called with the DCB as an argument.

```
Close   RTS
```

The StartIO routine is called with the DCB and the new packet as argument. The first thing to do is to extract the command type and see what is to be done. Besides the standard three functions of read, write, and reset, the disk device is also required to implement format, status, and motor off commands.

```
StartIO MOVE.L  8(SP),A0        Extract packet address
        MOVE.L  P_TYPE(A0),D0   Get packet action type
        MOVE.L  P_UNIT(A0),D1   ..and unit number
        LEA.L   UnitBase,A1     Set up pointer to
*                               data area
        MULU    #MaxData,D1     Make offset to unit
*                               data area
* Point to correct unit area
        ADDA.L  D1,A1
```

```
        CMP.L   #C_READ,D0      Check for valid types
        BEQ.S   Read
        CMP.L   #C_WRITE,D0
        BEQ     Write
        CMP.L   #C_RESET,D0
        BEQ     ResetAll
        CMP.L   #C_FORMAT,D0
        BEQ     DoFormat
        CMP.L   #C_STATUS,D0
        BEQ     DoStatus
        CMP.L   #C_MOTOR,D0
        BNE.S   Invalid
```

At this point you handle the motor timeout command. On this hardware you cannot actually turn the motor off, so instead you turn an LED off to signal that the disk may be removed from the drive. That done, you branch to a standard packet return code section. For example:

```
        BCLR    #4,MCR          Blank the display
        BRA.S   PktRet
```

If, on getting here, the packet contains an invalid action code, you return an error in the packet:

```
Invalid
        CLR.L   P_RES1(A0)              Set error result
        MOVE.L  #E_UT,P_RES2(A0)        Error code
```

Now you return the packet. Packets may be returned in any of these routines except the interrupt routine. You must unlink the packet from the work queue in the DCB and then call a subroutine to return the packet. You then see if there is another packet waiting at the head of the work queue; if there is, you set it to be the packet parameter and loop back to the start again:

```
PktRet   MOVE.L  4(SP),A1          Extract DCB address
         MOVE.L  (A0),(A1)         Unlink pkt from work Q
         CMPA.L  D_Tail(A1),A0     Check if last packet
         BNE.S   5$                No, more to come
         MOVE.L  A1,D_Tail(A1)     Set initial tail ptr
5$       BSR     Return
TryNext  TST.L   (A1)
         BEQ.S   6$
         MOVE.L  (A1),8(SP)
         BRA     StartIO
6$       RTS
```

At this point you handle a read or write request. As the two commands are identical apart from one byte, you can patch the command area accordingly:

```
Read     MOVE.B  #$10,ActD+4(A1)   Patch to be READ
         BRA.S   DoIt

Write    MOVE.B  #$20,ActD+4(A1)   Patch to be WRITE
```

Now insert the buffer address, the number of 512-byte data blocks, and the block number obtained from the byte offset specified in the packet. When this is done, code a jump to the SendC routine, which then sends the command and returns.

```
DoIt     MOVEM.L P_BUF(A0),D1-D3   Get buf, size, offset
         LEA.L   ActD(A1),A0       Point to command
         ORI.L   #$3D000000,D1     Add in memory modifier
         MOVE.L  D1,10(A0)         Update command area
         MOVEQ   #9,D1
         ASR.L   D1,D2             = # 512 byte blocks
         MOVE.W  D2,6(A0)          Set no. of blocks
         ASR.L   D1,D3             Into block offset
         MOVE.L  D3,16(A0)         And slot that in
         BSET    #4,MCR            Turn display on
         BRA     SendC
```

Here you must handle the format command. The arguments are identical to the Read and Write commands and, again, jump to the SendC routine to send the command to the controller:

```
DoFormat
        MOVE.L  P_OFF(A0),D3      Fetch Offset
        MOVEQ   #9,D1
        ASR.L   D1,D3             Into block offset
        LEA.L   FormD(A1),A0      Load command buffer
        MOVE.L  D3,10(A0)         And slot that in
        BSET    #4,MCR            Turn display on
        BRA     SendC
```

At this point you must implement the 'return device status.' In an ideal world, this packet should be returned only when the disk status changes (that is, on a door interrupt). However, this hardware, along with many others, does not implement door open and close interrupts. You must, therefore, test the current state and return the packet at once. The packet is only sent again after a suitable delay of three seconds or so. The status packet contains no arguments, and returns one of three values. Zero indicates that no disk is present in the drive. A value of one indicates that a write protected disk is inserted, and a value of two indicates a read/write disk.

```
DoStatus
        LEA.L   StatD(A1),A0      Issue command
        BRA     SendC
```

This standard subroutine is called from Open and also when the Reset command is received.

```
ResetAll
        LEA.L   InitD(A1),A0      Initialize the device
        BRA     SendC             Do it
```

This entry point returns the packet held in A0, and returns the value TRUE.

```
Return  MOVE.L  A0,D1          Return the packet
        MOVE.L  #-1,(A0)       Set notinuse
        MOVE.L  Devid,D2
        MOVEQ   #K_QPkt,D0
        TRAP    #0
        MOVEQ   #-1,D0
        RTS
```

The AbortIO call is called with the DCB and a packet address as arguments. It would normally only be used when a private queue of packets was being maintained, and then you would unchain the packet from this private queue. In this case, however, there is no work to be done.

```
AbortIO RTS                    Nothing to do
```

The system calls the RecallIO routine after an Interrupt call has indicated that this is needed. The call is normally only scheduled You when there is a packet to be moved back to a client. RecallIO is called with the DCB as the first argument, and the address of the Interrupt Transfer Block (ITB) that was associated with the interrupt. In this case, you know the packet that contains the command that caused the interrupt is at the head of the DCB work queue, so the first thing to do is to remove it from this queue:

```
RecallIO
        MOVE.L  4(SP),A1          Extract DCB address
        MOVEA.L (A1),A0           Get packet address
        MOVE.L  (A0),(A1)         Unlink pkt from work Q
        CMPA.L  D_Tail(A1),A0     Check if last packet
        BNE.S   7$                No, more to come
        MOVE.L  A1,D_Tail(A1)     Set initial tail ptr
```

The next stage is to see if the command worked. In the case of this hardware, you extract the status code and insert it into the result field. If the result indicates an error, you branch onwards. Firstly, however, a check is made to find out whether the current request is a 'status' command since this command is handled specially.

```
7$      CMPI.B  #$72,IDC_STTYPE See if STATUS request
        BNE.S   6$              No, so carry on
        MOVEQ   #0,D1           Assume not ready
        MOVE.B  IDC_STATUS,D0   Get status byte
        BTST    #7,D0           Check ready
        BNE.S   5$              Drive not ready
        MOVEQ   #1,D1           Assume write protected
        BTST    #5,D0           Check if so
        BNE.S   5$              Bit set so protected
        MOVEQ   #2,D1           Value is in D1
5$      MOVE.L  D1,P_RES1(A0)   Set result
        BRA.S   9$
```

Otherwise you return the size requested to indicate no error, and then, in all cases, you jump back to the shared code to return the packet. It is only at this point that you set the hardware to allow further interrupts. If you allowed further interrupts any earlier, Tripos would get confused with another recall queued up behind this one, both attempting to take the same packet off the head of the packet queue.

```
6$      MOVEQ.L #0,D0
        LEA.L   IDC_STC,A1
        MOVEP.W 0(A1),D0        Get two status codes
        MOVE.L  D0,P_RES2(A0)   Set secondary result
        BEQ.S   8$              No error
        CLR.L   P_RES1(A0)      Flag error
        BRA.S   9$              And return pkt
8$      MOVE.L  P_SIZE(A0),P_RES1(A0)
*                               Return length requested
9$      CLR.B   IDC_MSGSENT     All done, allow further
*                               interrupts
        BRA     Return

7$      MOVEQ.L #0,D0
        LEA.L   IDC_STC,A1
        MOVEP.W 0(A1),D0        Get two status codes
        MOVE.L  D0,P_RES2(A0)   Set secondary result
        BEQ.S   6$              No error
        CLR.L   P_RES1(A0)      Flag error
        BRA.S   9$              And return pkt
```

The final entry point is the Interrupt routine. This is called with the DCB and the ITB as arguments. As this is actually called as an interrupt, as little work as possible should be done here. The system uses the value returned by Interrupt to determine what should be done next. If the Interrupt routine returns a negative value, then this interrupt routine could not handle the interrupt, so it must have been for another device using the same interrupt slot. The system then calls the next routine in the chain. If the value returned is zero, then the interrupt was for this device, but no recall is required. This is because either there is no packet waiting on the queue, or because the interrupt routine has done all the work required at this point. A positive value returned indicates that a packet is to be moved, and so a later call of recall is requested. Normally the interrupt routine must turn off the interrupt; in this example the interrupt is turned off as soon as it is acknowledged by the hardware. The interruption is very short.

```
Int     MOVEA.L 4(SP),A0      Get DCB
        MOVEQ   #1,D0         Assume pkt present
        TST.L   D_Head(A0)    Check if true
        BNE.S   5$            Yes
        CLR.B   IDC_MSGSENT   All done, allow more ints
        MOVEQ   #0,D0         .. no, don't recall
5$      RTS
```

The rest of the driver is merely a subroutine and data areas. This subroutine sends the command that A0 points at, and requests that an interrupt be given when the command has completed. Most of the details are hardware specific.

```
SendC   TST.B   IDC_CMDSENT      Check last command sent
        BNE.S   SendC            Wait until swallowed
        MOVE.L  A3,-(SP)         Save A3
        LEA.L   IDC_CMDBUF,A3    Point to command buffer
        CLR.W   D0
        MOVE.B  2(A0),D0         Get size of command
        BRA.S   2$               Jump to end of loop
* Send the bytes of the command
1$      MOVE.B  (A0)+,(A3)
        ADDQ.L  #2,A3
2$      DBRA    D0,1$
        MOVE.B  #$80,IDC_CMDSENT Flag we sent it
        MOVE.L  (SP)+,A3
        RTS
```

Finally the data area, which contains the control function codes to be sent to the controller via the SendC subroutine, plus the device id location.

```
        DATA
UnitBase

* Unit 0 is the floppy disk

Unit0   EQU     *

InitFDC DC.B    $02,$00,$1D,$06,$40,$08,$32,$01
        DC.B    $08,$02,$02,$00,$00,$50,$00,$00
        DC.B    $00,$00,$00,$00,$01,$00,$00,$00
        DC.B    $00,$00,$00,$00,$03
        CNOP    0,2
ActF    DC.B    $02,$00,$15,$06,$10,$01,$00,$00
        DC.B    $02,$00,$3D,$00,$00,$00,$00,$00
        DC.B    $00,$00,$00,$00,$03
        CNOP    0,2
FormF   DC.B    $02,$00,$07,$06,$40,$02,$03,$00
        DC.B    $00,$00,$00,$00,$00,$00,$00
        CNOP    0,2
StatF   DC.B    $02,$00,$07,$06,$30,$02,$03
        CNOP    0,2
```

```
* Unit 1 will be the hard disc

Unit1   DC.L    0                   Pkt queue
*
InitHDC DC.B    $02,$00,$1D,$00,$40,$08,$00,$00
        DC.B    $20,$04,$01,$00,$01,$68,$00,$80
        DC.B    $01,$69,$0B,$00,$0D,$00,$00,$00
        DC.B    $00,$00,$00,$00,$03
        CNOP    0,2
ActH    DC.B    $02,$00,$15,$00,$10,$01,$00,$00
        DC.B    $02,$00,$3D,$00,$00,$00,$00,$00
        DC.B    $00,$00,$00,$00,$03
        CNOP    0,2
FormH   DC.B    $02,$00,$0F,$00,$40,$01,$00,$08
        DC.B    $02,$00,$00,$00,$00,$00,$03
        CNOP    0,2
StatH   DC.B    $02,$00,$07,$00,$30,$02,$03
        CNOP    0,2

Devid   DC.L    0
*
* General commands to the IDC
*
InitIDC DC.B    $02,$00,$1D,$00,$50,$00,$40,$80
        DC.B    $00,$00,$00,$00,$00,$00,$00,$00
        DC.B    $00,$00,$00,$00,$00,$00,$00,$00
        DC.B    $00,$00,$00,$00,$03,$00

        END
```

Serial Device Driver

The next example is of a slightly more complicated device, being a driver for a serial device. This supports two different units. Firstly the DCB, which is similar to the disk example but which contains four ITBs. You might at first think that there would be only two ITBs as there are two interrupts from this device; one for each unit. But although there is no difference between a reception interrupt and a transmission interrupt it is convenient to imagine that there is.

The driver has been written so that the two cases of read (reception) and write (transmission) are distinct. If an interrupt occurs the read entry point will be called via the ITB with the value IntR. If it was a read interrupt then it will be handled. If it was instead a write interrupt then the IntR interrupt routine will return a negative value, indicating that the interrupt could not be handled. The kernel will then use the next ITB in the chain for this interrupt vector to see if can be handled there. This will be the ITB with the value IntW for the interrupt routine, which should now be able to cope.

The user data area of the ITB is used to point to the unit data structure in the driver. This area contains different packet queues for the different units, and also contains the device address in memory. Thus each ITB has a different combination of IntR, IntW, Unit0 and Unit1.

```
        XREF    Open
        XREF    Close
        XREF    StartIO
        XREF    AbortIO
        XREF    RecallIO
        XREF    IntR,IntW
        XREF    Unit0,Unit1

Base    DC.L     0          Work Queue Head
        DC.L     Base       Work Queue Tail
        DC.L     0          Qpkt Action routine
        DC.L     0          DQpkt Action routine
        DC.L    10          Queue Priority
        DC.L     0          Device identifier

* Links to entry points
        DC.L     Open
        DC.L     Close
        DC.L     StartIO
        DC.L     AbortIO
        DC.L     RecallIO

* Interrupts used

        DC.L     4          Number of ITBs
```

```
* Unit 0 Write
        DS.W    3       (Used by kernel)
        DC.W    Base-*  Offset to DCB
        DC.L    0       Pointer to next ITB
        DC.L    IntW    Interrupt routine
        DC.L    $70     Interrupt offset
        DC.L    0       Recall Q
        DC.L    Unit0   User data area
        DC.L    0       Packet to be returned

* Unit 0 Read
        DS.W    3       (Used by kernel)
        DC.W    Base-*  Offset to DCB
        DC.L    0       Pointer to next ITB
        DC.L    IntR    Interrupt routine
        DC.L    $70     Interrupt offset
        DC.L    0       Recall Q
        DC.L    Unit0   User data area
        DC.L    0       Packet to be returned

* Unit 1 Write
        DS.W    3       (Used by kernel)
        DC.W    Base-*  Offset to DCB
        DC.L    0       Pointer to next ITB
        DC.L    IntW    Interrupt routine
        DC.L    $74     Interrupt offset
        DC.L    0       Recall Q
        DC.L    Unit1   User data area
        DC.L    0       Packet to be returned

* Unit 1 Read
        DS.W    3       (Used by kernel)
        DC.W    Base-*  Offset to DCB
        DC.L    0       Pointer to next ITB
        DC.L    IntR    Interrupt routine
        DC.L    $74     Interrupt offset
        DC.L    0       Recall Q
        DC.L    Unit1   User data area
        DC.L    0       Packet to be returned
```

The serial device driver is presented here. The majority of the code is identical to that used in the disk driver, so little extra commentary is added. Note, however, the two commands specific to the terminal driver - GPARMS & SPARMS.

```
        INCLUDE "tripos.i"
* Standard linkage

        XDEF    Open
        XDEF    Close
        XDEF    StartIO
        XDEF    AbortIO
        XDEF    RecallIO
        XDEF    IntR,IntW
        XDEF    Unit0,Unit1

* Packet offests

P_UNIT  EQU     P_ARG1
P_BUF   EQU     P_ARG2
P_SIZE  EQU     P_ARG3

* TTY commands

C_SPARMS EQU    8
C_GPARMS EQU    9

* Parameter buffer offsets

A_SPEED EQU     0
A_BPW   EQU     4
A_STOPBITS EQU  8
A_PARITY EQU    12

* Standard constants

E_UT    EQU     12      Unkown type error
```

```
* Private data area structure

* Queue of read packets not yet handled
ReadQ     EQU     0
* Queue of write packets not yet handled
WriteQ    EQU     8
ReadB     EQU    16  Read buffer base
ReadL     EQU    20  Read buffer limit
WriteB    EQU    24  Write buffer current base
WriteL    EQU    28  Write buffer limit
Device    EQU    32  Device register base
Speed     EQU    36
Bpw       EQU    40
StopBits  EQU    44
Parity    EQU    48

UnitDat   EQU    32  Uninitialised data size
MaxData   EQU    52  Maximum data size per unit
MaxUnit   EQU     2  Maximum number of units supported

* Device register offsets

DReg      EQU      0
SReg      EQU      2
MReg      EQU      4
CReg      EQU      6

* Initialistaion values

Clockrate EQU  $3E   9600 baud, Internal clocks
IStopBits EQU  $4E   1 stop bit, 8 bits, async 16x
InitCR    EQU  $24   Normal mode, RTS active, DTR
*                    inactive, TxEN*, RxEN
```

```
******************
* The Open routine
* Open( dcb )
******************

Open    MOVEA.L 4(SP),A1        Get DCB ptr
        MOVE.L  D_Id(A1),Devid  Save my device ID
* Now Reset all the devices
        LEA.L   Unit0,A1
        BSR     ResetAll
        LEA.L   Unit1,A1
        BSR     ResetAll
        RTS

******************
* The Close routine
* Close( dcb )
******************
Close   RTS

*****************************************************
* The StartIO routine. Called with the dcb and packet
* as arguments
* StartIO( dcb, pkt )
*****************************************************
StartIO MOVE.L  8(SP),A0        Extract packet address
        MOVE.L  4(SP),A1        Extract DCB address
        MOVE.L  (A0),(A1)       Unlink pkt from work Q
        CMPA.L  D_Tail(A1),A0   Check if last packet
        BNE.S   5$              No, more to come
        MOVE.L  A1,D_Tail(A1)   Set initial tail ptr
5$      MOVE.L  P_TYPE(A0),D0   Get packet action type
        MOVE.L  P_UNIT(A0),D1   ..and unit number
        LEA.L   UnitBase,A1     Set up pointer to data
*                               area
        MULU    #MaxData,D1     Make offset to unit
*                               data area
        ADDA.L  D1,A1           Point to correct unit
*                               area
```

```
        CMP.L   #C_READ,D0      Check for valid types
        BEQ.S   Read

        CMP.L   #C_WRITE,D0
        BEQ     Write

        CMP.L   #C_RESET,D0
        BEQ     ResetAll

        CMP.L   #C_GPARMS,D0
        BEQ     GParms

        CMP.L   #C_SPARMS,D0
        BEQ     SParms

* Invalid action request

        CLR.L   P_RES1(A0)          Set error result
        MOVE.L  #E_UT,P_RES2(A0)    Error code
        BRA     Return              Return the packet

* Handle a read request

Read    MOVE.L  A2,-(SP)
        TST.L   ReadQ(A1)       Will it be the first ?
        BNE.S   1$              No so just q it
        LEA.L   ReadQ(A1),A2    Get queue ptr
        BSR.S   Add2Q
        BSR.S   ReadInit
        BRA.S   2$
* Slot this packet onto an internal queue until the
* request can be handled

1$      LEA.L   ReadQ(A1),A2    Get queue ptr
        BSR.S   Add2Q
2$      MOVEA.L (SP)+,A2
        RTS
```

```
ReadInit
        MOVE.L  P_BUF(A0),A2    Load buffer pointer
        MOVE.L  P_SIZE(A0),D0   Fetch size
        BEQ     WorkDone
        ADD.L   A2,D0
        MOVE.L  D0,ReadL(A1)    Set up buffer end
        MOVE.L  A2,ReadB(A1)    Set up read buffer
        MOVEA.L Device(A1),A2
        BSET    #1,CReg(A2)     DTR active
        RTS
*
Add2Q   MOVE.L  A3,-(SP)
        MOVE.L  4(A2),A3        Fetch tail pointer
        MOVE.L  A0,(A3)         Add pkt to end of Q
        CLR.L   (A0)            Set as end
        MOVE.L  A0,4(A2)        Save in tail slot
        MOVE.L  (SP)+,A3
        RTS                     And exit
*
TakeQ   MOVE.L  (A2),A0
        MOVE.L  (A0),(A2)
        BNE.S   1$
        MOVE.L  A2,4(A2)
1$      RTS

* Handle a Write request
* A1 = Unit data area
*  Corrupted A0/A1/D0/D1
*
Write   MOVE.L  A2,-(SP)        Save A2
        TST.L   WriteQ(A1)      Will it be head
        BNE.S   1$
        LEA.L   WriteQ(A1),A2   Point to queue head
        BSR.S   Add2Q           Add to the queue,
*                               restore A2
        BSR.S   WriteInit
        BRA.S   2$
*
* Add the packet to the internal write q
*
```

```
1$      LEA.L   WriteQ(A1),A2    Point to queue head
        BSR.S   Add2Q             Add to the queue,
*                                 restore A2
2$      MOVEA.L (SP)+,A2
        RTS
*
WriteInit
        MOVE.L  P_BUF(A0),A2    Load buffer pointer
        MOVE.L  P_SIZE(A0),D0   Fetch size
        BEQ     WorkDone
        ADD.L   A2,D0
        MOVE.L  D0,WriteL(A1)   Set up buffer end
        MOVE.B  (A2)+,D0        Fetch first char
        MOVE.L  A2,WriteB(A1)   Save buffer storage area
* Turn TX interrupts on & handle the buffer.
        MOVEA.L Device(A1),A2   Fetch device register
*                               base
        MOVE.B  D0,DReg(A2)     Send char
        BSET    #0,CReg(A2)     Enable transmitter
        RTS
*
ResetAll MOVEA.L  Device(A1),A0
        TST.B   CReg(A0)              Point to MR1
        MOVE.B  #IStopBits,MReg(A0) Set control bits
        MOVE.B  #Clockrate,MReg(A0) Set baud rate
        MOVE.B  #InitCR,CReg(A0)
        RTS
*
* Get parameters command
*
GParms  MOVE.L  A2,-(SP)
        MOVE.L  P_BUF(A0),A2
* Fetch current speed
        MOVE.L  Speed(A1),(A2)+
* Fetch current bits per word
        MOVE.L  Bpw(A1),(A2)+
* Fetch current no. of stop bits
        MOVE.L  StopBits(A1),(A2)+
* Fetch current parity
        MOVE.L  Parity(A1),(A2)+
```

```
        MOVEA.L (SP)+,A2
        MOVE.L  #-1,P_RES1(A0)      TRUE result
        BRA     Return
*
* Set parameters command
*
SParms  MOVEM.L A2-A3,-(SP)         Save Regs
*  Fetch parameter buffer
        MOVE.L  P_BUF(A0),A2
        BSR     SP2
* Save as new speed
        MOVE.L  A_SPEED(A2),Speed(A1)
* Save as new bits per word
        MOVE.L  A_BPW(A2),Bpw(A1)
* Save as new no. of stop bits
        MOVE.L  A_STOPBITS(A2),StopBits(A1)
* Save as new parity
        MOVE.L  A_PARITY(A2),Parity(A1)
        MOVEM.L (SP)+,A2/A3
        BRA     Return
*
* SP2 sets EPCI parameters
*  Inputs A2 - Parameter buffer
*         A1 - Unit data area
*         A0 - Packet
*
*  Corrupted - D0/A3/D1
*
SP2     MOVE.L  A_SPEED(A2),D0      Fetch required speed
        CMP.L   Speed(A1),D0        Same as before ?
        BEQ.S   3$                  If EQ yes
        LEA.L   SpeedTab,A3         Load Speed table
1$      TST.W   (A3)                Any more ?
        BEQ.S   4$                  If EQ no
        CMP.W   (A3)+,D0            This one ?
        BEQ.S   2$                  If EQ yes
        ADDQ.L  #2,A3               Skip control bytes
        BRA.S   1$                  Round again
4$      MOVE.L  #0,P_RES1(A0)       Set fail
        RTS
```

```
2$      MOVE.W   (A3),D0            Fetch code
        MOVEA.L  Device(A1),A3      Load device address
        TST.B    CReg(A3)           Point to MR1
        TST.B    MReg(A3)           Point to MR2
        MOVE.B   MReg(A3),D1        Fetch MR2
        ANDI.B   #$F0,D1            Mask previous speed
        OR.B     D0,D1              Insert new speed
        TST.B    MReg(A3)           Point to MR2
        MOVE.B   D1,MReg(A3)        Set new speed
3$      MOVEA.L  Device(A1),A3      Ensure device pointer
        TST.B    CReg(A3)           Point to MR1
        MOVE.B   MReg(A3),D1        Fetch current MR1
        MOVE.L   A_BPW(A2),D0       Fetch required BPW
        CMP.L    Bpw(A1),D0         Same as before ?
        BEQ.S    5$                 If EQ yes
* Make appropriate value
        SUBQ.B   #5,D0
* Shift up to correct slot
        LSL.B    #2,D0
        ANDI.B   #$F3,D1            Remove previous value
        OR.B     D0,D1              Slot in new value
5$      MOVE.L   A_STOPBITS(A2),D0  Fetch no. of stopbits
        CMP.L    StopBits(A1),D0    Same as before ?
        BEQ.S    6$                 If EQ yes
* Make appropriate value
        ADDQ.B   #1,D0
        LSL.B    #6,D0              To correct slot
        ANDI.B   #$3F,D1            Mask previous value
        OR.B     D0,D1              Insert new value
6$      MOVE.L   A_PARITY(A2),D0    Fetch parity code
        CMP.L    Parity(A1),D0      Same as before
        BEQ.S    7$                 If EQ yes
        TST.B    D0                 No parity ?
        BEQ.S    61$                If EQ yes
        CMP.B    #1,D0              Odd parity ?
        BEQ.S    62$                If EQ yes
        MOVE.B   #$30,D0            Otherwise even parity
        BRA.S    61$
62$     MOVE.B   #$1,D0             Set odd parity value
61$     ANDI.B   #$CF,D1            Mask previous value
```

```
        LSL.B   #4,D0               To correct slot
        OR.B    D0,D1               Insert new value
7$      TST.B   CReg(A3)            Point to MR1
        MOVE.B  D1,MReg(A3)         Set new MR1
        MOVE.L  #-1,P_RES1(A0)
        RTS
*
* A0 = Packet
* Corrupted D0/D1/D2
*
WorkDone
        MOVE.L  P_SIZE(A0),P_RES1(A0)
*
* Return the packet in A0, and return the value TRUE.
* Corrupts D0/D1/D2
Return  MOVE.L  A0,D1               Return the packet
        MOVE.L  #-1,(A0)            Set notinuse
        MOVE.L  Devid,D2
        MOVEQ   #K_QPkt,D0
        TRAP    #0
        MOVEQ   #-1,D0
        RTS

*********************************************************
* The AbortIO call. Cancel the request packet specified
* as argument
* AbortIO( dcb, pkt )
*********************************************************
AbortIO MOVEA.L 8(SP),A0            Fetch packet
        MOVE.L  A2,-(SP)
        MOVE.L  P_UNIT(A0),D0
        LEA.L   UnitBase,A1
        MULU    #MaxData,D0
        ADDA.L  D0,A1
        LEA.L   ReadQ(A1),A2        Load read q
        CMPI.L  #C_READ,P_TYPE(A0)  Read command ?
        BEQ.S   1$                  If EQ yes
* Otherwise load write q
        LEA.L   WriteQ(A1),A2
1$      MOVE.L  A2,-(SP)            Save q head
```

```
3$      CMPA.L  (A2),A0             This packet ?
        BEQ.S   2$                  If EQ yes
        MOVEA.L (A2),A2             Otherwise chain on
        CMPA.L  #0,A2               Any more ?
        BNE.S   3$                  If NE yes
        ADDQ.L  #4,SP               Junk list head
* Otherwise restore A2
        MOVE.L  (SP)+,A2
        RTS                         And exit
2$      MOVE.L  (A0),(A2)           Unlink
        MOVE.L  (SP)+,A0            Restore q head
        TST.L   (A2)                Last packet ?
        BNE.S   9$                  If NE no
        MOVE.L  A2,4(A0)            Set new end pointer
5$      TST.L   (A0)                Anything left on q
        BNE.S   9$                  If NE yes
        MOVE.L  12(SP),A0           Fetch packet again
        CMPI.L  #C_READ,P_TYPE(A0)  Read command?
        BEQ.S   8$                  If EQ yes
        CLR.L   WriteB(A1)          Set no write buffer
        BRA.S   6$
8$      CLR.L   ReadB(A1)           Set no read buffer
        BRA.S   6$                  And exit
*
9$      CMPA.L  A2,A0               Was packet at head?
        BNE.S   6$                  If NE no
* Otherwise fetch new head packet
        MOVE.L  (A0),A0
        CMPI.L  #C_READ,P_TYPE(A0)  Read command?
        BEQ.S   7$                  If EQ yes
* Otherwise issue new write
        BSR     WriteInit
        BRA.S   6$
7$      BSR     ReadInit            Issue new read
6$      MOVE.L  (SP)+,A2
        RTS                         And exit
```

```
*****************************************************
* The Recall Routine. This entry point is called when
* there is a packet to be moved back to a client.
* Packets to be returned are held on another queue
* within the unit storage area
* RecallIO( dcb, itb )
*****************************************************
RecallIO
        MOVEA.L 8(SP),A1              Fetch ITB
        MOVEA.L ITB_PKT(A1),A0        Fetch packet
        MOVEA.L ITB_USR(A1),A1        Fetch unitbase
        CMP.L   #C_READ,P_TYPE(A0)    Read action
        BNE.S   2$                    If NE no
        BSR     WorkDone              Return pkt
        TST.L   ReadQ(A1)
        BEQ.S   1$
        MOVE.L  ReadQ(A1),A0
        MOVE.L  A2,-(SP)
        BSR     ReadInit
        MOVEA.L (SP)+,A2
        BRA.S   3$
1$      CLR.L   ReadB(A1)           DTR already inactive
        MOVE.L  Device(A1),A0
3$      RTS
*
2$      BSR     WorkDone
        TST.L   WriteQ(A1)
        BEQ.S   4$
        MOVE.L  WriteQ(A1),A0
        MOVE.L  A2,-(SP)
        BSR     WriteInit
        MOVEA.L (SP)+,A2
        BRA.S   5$
4$      CLR.L   WriteB(A1)
5$      RTS
```

```
**************************************************
* The interrupt routine
* res = Int( dcb, itb )
*
* res < 0  -> Interrupt not handled
* res = 0  -> Interrupt handled, do not recall
* res > 0  -> Interrupt handled, recall me
**************************************************
*
* IntR handles read requests
*
IntR    MOVEQ.L #1,D0           Status bit to test
        LEA.L   ReadI,A0
        BRA.S   Dispatch
*
* IntW handles write requests
*
IntW    MOVEQ.L #0,D0           Status bit to test
        LEA.L   WriteI,A0

Dispatch
        MOVEA.L 8(SP),A1        Fetch ITB
        MOVEM.L A2/A3,-(SP)     Save regs
        MOVEA.L ITB_USR(A1),A1  Fetch unitbase
        MOVEA.L Device(A1),A3   Get pointer to data port
        BTST    D0,SReg(A3)     Handlable by this
*                               routine?
        BNE.S   1$              If NE yes
        MOVEM.L (SP)+,A2/A3     Otherwise restore regs
        MOVEQ.L #-1,D0          And reject interrupt
        RTS
1$      JMP     (A0)            Call correct routine

* Handle Read interrupt

ReadI   MOVE.B  DReg(A3),D0     Read to cancel interrupt
        BCLR    #1,CReg(A3)     DTR inactive
        MOVEA.L ReadB(A1),A2
        CMPA.L  #0,A2           Buffer present?
        BEQ.S   IntE            No recall no work
```

```
        MOVE.B  D0,(A2)+        Copy byte
        CMPA.L  ReadL(A1),A2    End of buffer?
        BEQ.S   3$
        MOVE.L  A2,ReadB(A1)    Update buffer pointer
        BRA.S   IntE            No recall
3$
        LEA.L   ReadQ(A1),A2
        BSR     TakeQ           Get waiting packet
*
IntO    MOVEM.L (SP)+,A2/A3     Restore A2/A3
        MOVEA.L 8(SP),A1
        MOVE.L  A0,D0
        MOVE.L  D0,ITB_PKT(A1) Ask to be recalled
        RTS

* Exit used when no Recall required
IntE    MOVEM.L (SP)+,A2/A3     Restore A2/A3
        MOVEQ   #0,D0
        RTS


* Handle Write interrupt

WriteI  MOVEA.L WriteB(A1),A2  Get buffer ptr
        CMPA.L  #0,A2           Buffer ready?
        BEQ.S   IntE            No recall, no work
        CMPA.L  WriteL(A1),A2  Any more to do?
        BLT.S   5$              Yes, do it
        BCLR    #0,CReg(A3)     Disable transmitter
* Fetch packet head packet
        LEA.L   WriteQ(A1),A2
        BSR     TakeQ           Unlink it
        BRA.S   IntO
* Place character into port
5$      MOVE.B  (A2)+,DReg(A3)
        MOVE.L  A2,WriteB(A1)  Replace buffer ptr
        BRA.S   IntE
        PAGE
```

```
* Data areas used by the driver

*         DATA
UnitBase
*
Unit0    DC.L    0,*
         DC.L    0,*
         DS.B    UnitDat-16
         DC.L    $FE00A1          USART control port
         DC.L    9600,8,0,0
*
Unit1    DC.L    0,*
         DC.L    0,*
         DS.B    UnitDat-16
         DC.L    $FE00B1          USART control port
         DC.L    9600,8,0,0
*
Devid    DC.L    0
*
SpeedTab DC.W    19200,15
         DC.W    9600,14
         DC.W    7200,13
         DC.W    4800,12
         DC.W    3600,11
         DC.W    2400,10
         DC.W    2000,9
         DC.W    1800,8
         DC.W    1200,7
         DC.W    600,6
         DC.W    300,5
         DC.W    150,4
         DC.W    110,2
         DC.W    75,1
         DC.W    50,0
         DC.W    0,0          Terminate table
         END
```

Clock Device Driver

The final example is the clock device. This is a slightly special case because the action of the timer is different from the others. Firstly it only copes with one special packet which contains a number of ticks. When the specified number of ticks have expired the packet is returned. Secondly at every tick (20ms) the device is interrupted and the time, stored in the root node, is updated. The device is implemented as a standard device, but maintains a private packet queue in which the packets are ordered by their expiration time. The DCB is fairly standard, as follows.

```
        XREF    Open
        XREF    Close
        XREF    StartIO
        XREF    AbortIO
        XREF    RecallIO
        XREF    Int

Base    DC.L    0       Work Queue Head
        DC.L    Base    Work Queue Tail
        DC.L    0       Qpkt Action routine
        DC.L    0       DQpkt Action routine
        DC.L    8       Queue Priority
        DC.L    0       Device identifier

* Links to entry points

        DC.L    Open
        DC.L    Close
        DC.L    StartIO
        DC.L    AbortIO
        DC.L    RecallIO
```

```
* Interrupts used

        DC.L    1       Number of ITB

        DS.W    3       (Used by kernel)
        DC.W    Base-*  Offset to DCB
        DC.L    0       Pointer to next ITB
        DC.L    Int     Interrupt routine
        DC.L    $68     Interrupt offset
        DC.L    0       Recall Q
        DC.L    0       User data area
        DC.L    0       Packet to be returned
```

The device driver code itself is reasonably simple. An extra header must be included, and the constant section describes the hardware used here.

```
        INCLUDE "tripos.i"
        INCLUDE "mchdr.i"

* Standard linkage

        XDEF    Open
        XDEF    Close
        XDEF    StartIO
        XDEF    AbortIO
        XDEF    RecallIO
        XDEF    Int

* Packet offsets

P_TICKS EQU     P_ARG1

* Standard constants

E_UT    EQU     12      Unkown type error
```

```
* Device registers

PTM         EQU         $FE00D1
CLKA        EQU         PTM+$0
CLKB        EQU         PTM+$2
CLKC        EQU         PTM+$0        (same as CLKA)
CLKMSB      EQU         PTM+$4
CLKLSB      EQU         PTM+$6
C3LATCH     EQU         PTM+$E
C3COUNTER   EQU         PTM+$C
CLKSTATUS   EQU         PTM+$2        (same as CLKB)

MTICKS      EQU         50*60
```

The Open routine must initialize the clock, which is merely a tedious bit-twiddling exercise with this hardware.

```
Open        MOVE.L   4(SP),A1
            MOVE.L   D_Id(A1),Devid
            MOVE.B   #$13,CLKMSB     5000 @ 2MHz/8
            MOVE.B   #$88,C3LATCH    ($1388 = 5000)
            MOVE.B   #0,CLKB         Point to Clock 3
            MOVE.B   #$C1,CLKC       Enable, external
            MOVE.B   #$01,CLKB       Disable clock2
            MOVE.B   #$00,CLKA       Disable clock1
            RTS
```

The Close routine has almost nothing to do.

```
Close       MOVE.B   #0,CLKB
            MOVE.B   #0,CLKC         Disable clock3
            RTS
```

The StartIO routine will be called for each packet, as the routine always takes the packet off the DCB work queue and stores the packet on its own private work queue. This private work queue is ordered by the expiry time of the packets. Each packet has stored in the RES1 field the extra time left after the predecessor in the queue expires.

```
StartIO   MOVE.L  A2/D4,-(SP)      Save A2
          MOVE.L  16(SP),A0        Fetch packet
          MOVE.L  12(SP),A1        Fetch DCB
          MOVE.L  (A0),D_Head(A1)
          CMP.L   D_Tail(A1),A0
          BNE.S   5$
          MOVE.L  A1,D_Tail(A1)
5$        MOVE.L  P_TICKS(A0),D4 D4 = tick count
          BLE.S   QPCLK4         Delay <=0 return pkt
          LEA.L   TimerQ,A1      A1 = addr of first
*                                WKQ link word
*
QPCLK1    MOVE.L  A1,A2          A2 = addr of last
*                                link word
          MOVEA.L (A1),A1        Chain down one
          MOVE.L  A1,D0
          BEQ.S   QPCLK3         J if at end of list
*
          SUB.L   P_RES1(A1),D4 Subtract packet's
*                                ticks
          BGE.S   QPCLK1         J if ticks to go is
*                                still >=0
*
          ADD.L   P_RES1(A1),D4  Add ticks back in
*
* Prepare to insert the clock packet at this point
* in the list by correcting the ticks count of the
* packet that is here.
*
          SUB.L   D4,P_RES1(A1)  Correct ticks
*                                count of next
*                                packet
*
QPCLK3    MOVE.L  D4,P_RES1(A0)  Plant tick count
          MOVE.L  (A2),(A0)      Link packet in
          MOVE.L  A0,(A2)
*
Ret       MOVEA.L (SP)+,A2/D4    Restore A2
          MOVE.L  #-1,D0
          RTS
```

```
*
* Return packet with negative count value
*
QPCLK4      BSR      SendIt
            MOVE.L   #-1,D0          Res of QPKT
            BRA.S    Ret

SendIt      MOVE.L   D4,P_RES1(A0)   Plant calculated
*                                    ticks count
            MOVE.L   #-1,(A0)
            MOVE.L   A0,D1
            MOVE.L   Devid,D2
            MOVEQ    #K_QPkt,D0
            TRAP     #0
            RTS
```

The AbortIO call must search the private work queue for the packet and unhook it, and then adjust the time interval held in the next packet in the queue.

```
AbortIO     MOVE.L   A2,-(SP)
            MOVE.L   12(SP),A1
            LEA.L    TimerQ,A0
2$          TST.L    (A0)
            BEQ.S    3$
            CMPA.L   (A0),A1
            BEQ.S    1$
            MOVEA.L  (A0),A0
            BRA.S    2$
```

```
1$          MOVE.L   (A1),A2         Unlink it
            MOVE.L   A2,(A0)         A2 = ptr to next
*                                    clock packet
            BEQ.S    3$              J if no clock pkt to
*                                    correct
            MOVE.L   P_RES1(A1),D0   Correct the
            ADD.L    D0,P_RES1(A2)   ticks to go field in
*                                    the next packet
3$          MOVE.L   (SP)+,A2
            RTS
```

The RecallIO routine is scheduled when a packet must be moved back. The packet to be returned is the packet at the head of the private queue, and possibly others at the head of the queue.

```
RecallIO    MOVE.L   TimerQ,A0                Load packet
1$          MOVE.L   (A0),TimerQ              Unlink
            MOVE.L   P_TICKS(A0),P_RES1(A0)   Set result
            BSR      SendIt            Return packet
            MOVE.L   TimerQ,D0         Load next packet
            BEQ.S    2$
            MOVEA.L  D0,A0
            TST.L    P_RES1(A0)        Return this too ?
            BEQ.S    1$                If EQ, yes
            RTS                        Otherwise exit
2$
```

The interrupt routine must update the time of day fields held in the rootnode. It must also adjust the ticks value held in the first packet on the queue. If any packet is about to expire, then the RecallIO routine is requested.

```
Int         TST.B    CLKSTATUS
            TST.B    C3COUNTER
            TST.B    CLKLSB            Clear Interrupt
```

```
          LEA.L     ROOTNODE+TICKS,A0 R = MC addr of
*                                     timer words
          ADDQ.L    #1,(A0)           Inc TICKS
          CMPI.L    #MTICKS,(A0)      End of minute?
          BLT.S     CLOCK1            No
          CLR.L     (A0)              Yes, reset TICKS
          ADDQ.L    #1,-(A0)          Inc MINS
          CMPI.L    #60*24,(A0)       End of day?
          BLT.S     CLOCK1            No
          CLR.L     (A0)              Clear the mins
          ADDQ.L    #1,-(A0)          Inc days

CLOCK1    MOVE.L    TimerQ,A0         Fetch head of q
          MOVE.L    A0,D0             Is there one
          BEQ.S     NoRec             If EQ, no
          SUB.L     #1,P_RES1(A0)
          BEQ.S     Rec
NoRec     MOVEQ.L   #0,D0             No recall
          RTS
Rec       MOVEQ.L   #1,D0             Recall
          RTS
```

Finally the data area, which merely holds the device id and the head of the private packet queue.

```
          DATA
TimerQ    DC.L      0
Devid     DC.L      0
```

4.5.4 Device Dependent Library

Part of the implementation-specific details of any Tripos port is the device-dependent library routines. These library routines constitute a set of functions that must be implemented for a particular piece of hardware (for example, the getting and setting of a real time-of-day clock, which may be a null operation if the hardware does not exist) The library also has the ability to write a panic message, which is used if the system has some sort of catastrophic error while it is trying to initialize itself.

The example here shows what is required. The definitions required are as follows.

a) Linkage to kernel via rootnode
b) Initialization required
c) Level 7 interrupt handling
d) Reboot service
e) Panic message service
f) Stand alone I/O
g) Real Time Clock service

The items a to e are called via known locations in absolute store, while items f and g are assembler subroutines which may only corrupt D0-D7 and A3-A4. The example starts with includes and equates.

```
          INCLUDE "tripos.i"
          INCLUDE "mchdr.i"
*
* Equates
*
* Chip addresses and device specific values
*
EPCI      EQU     $FE00A0
E_RWD     EQU     1
E_SR      EQU     3
E_MR      EQU     5
E_CR      EQU     7
MCR       EQU     $FE00F1
PROM      EQU     $F00000
```

```
*
* Vector offsets
*
IV_BERR    EQU     $8     Bus error interrupt vector
INTSLOT1   EQU     $64    Level 1 autovector
INTSLOT2   EQU     $68
INTSLOT3   EQU     $6C
INTSLOT4   EQU     $70
INTSLOT5   EQU     $74
INTSLOT6   EQU     $78
INTSLOT7   EQU     $7C    Level 7 autovector
USERSLOT   EQU     $100    User interrupt vector start
*
* Globals called
*
G_STIME    EQU     6
G_RTIME    EQU     7
G_SARDCH   EQU     21
G_SAWRCH   EQU     22
MTICKS     EQU     50*60
```

The next section describes the absolute store required. The value of ROOTNODE is defined in the header mchdr.i and is the location of the rootnode. The initialization, reboot request and panic message writing is done here.

```
        ORG.L   ROOTNODE+CLKINIT
        DC.L    TriposINIT
        DC.L    BOOTREQ
        DC.L    PANICREQ
```

The next section is the start of the relocatable store. The code starts with some magic that makes this module look like a BCPL assembler module. If you change it, leave the same number of characters in the string. The actual code is not displayed here.

The first job to be done is the system initialization. This handles any initialization required which is not done by device drivers. Examples may include setting up default interrupt vectors, initializing interrupt priority encoders and so on. If you wish to print a message here then this

is the place to do it. The unused interrupt entries have the value stored at DEVUNSET placed into them. Entries which are going to be used are cleared to zero. The level seven autovector, which is connected to an abort button on this hardware, is also set up here. The final job is to set an LED display to B so that the disk device can flag the disks as busy.

```
TriposINIT
          MOVE.L   DEVUNSET,D0 Set up default handlers
          LEA.L    INTSLOT1,L  .. for unset devices
          MOVEQ.L  #5,D1
L1        MOVE.L   D0,(L)+
          DBRA     D1,L1
* Set up level 7 autovector (ABORT button)
          LEA.L    INT7,A1
          MOVE.L   A1,(L)
* Clear interrupt slots used by standard devices
          CLR.L    INTSLOT2
          CLR.L    INTSLOT4
          CLR.L    INTSLOT5
          CLR.L    USERSLOT
* Set the display to B for busy
          MOVE.B   #$DB,MCR
          RTS
```

The next job is to provide support for the level seven interrupt used as an abort button. Simply jump back into the kernel if you want the system to handle it, but don't corrupt any registers.

```
INT7      MOVE.L   ROOTNODE+ABORTHAND,-(SP)
          RTS
```

The next entry point required is the reboot request, which can be called in either user mode or super mode. You would normally call the resident bootstrap ROM again. Note that a bootstrap call routine should be called in super mode with interrupts off.

```
BOOTREQ   LEA.L   1$,A0
          MOVE.L  A0,$84
          TRAP    #1            Ensure supervisor mode
1$        MOVE    #$2700,SR
          MOVE.L  PROM,SP
          MOVE.L  PROM+4,A0
          JMP     (A0)          And go there
```

The next section is more complicated, but need not normally be changed. The entry point is called if something has gone wrong and there is no debug task available yet. This is commonly the case when attempting to port Tripos. Once the boot process gets far enough for the resident debugger to work then you can use that to debug. Until then this entry point gets called. Register D1 is set pointing (as a BCPL pointer) to a debug packet, which is a memory area containing the reasons for failure and a register dump.

```
DBPTASK   EQU     4
DBPCODE   EQU     8
DBPSP     EQU     16
DBPREGS   EQU     36
*
PANICREQ  MOVE.L  D1,D7
          LEA.L   MESS1,L       Point to message
          BSR     WRITES
          MOVE.L  ROOTNODE+KSTART,D0 Print KSTART addr
          BSR     WRHEX
          LEA.L   MESS2,L
          BSR     WRITES
          ASL.L   #2,D7         Debug pkt as m/c addr
          MOVEA.L D7,L          Point to space
          MOVE.L  DBPTASK(L),D0 Get taskid
          BMI.S   PANIC3        If <0, registers in
*                               DBPREGS
          MOVEA.L DBPSP(L),B    If >0, saved on
*                               saved SP
          MOVE.L  B,D6          Save ptr
          ADD.L   #60,D6        SR & PC are 60
*                               bytes beyond
          BRA.S   PANIC4
```

```
PANIC3    LEA.L   DBPREGS(L),B  Point to saved
*                               register dump
* Note that SR and PC are saved on SP
          MOVE.L  DBPSP(L),D6
PANIC4    BSR     WRHEXSP       Write out task ID
          MOVE.L  DBPCODE(L),D0 Extract code
          BSR     WRHEXSP       .. and write
          MOVE.L  DBPSP(L),D0   Write out SP
          BSR     WRHEX
          LEA.L   MESS3,L
          BSR     WRITES
          MOVEQ   #14,D7        Counter for
*                               15 registers
PANIC5    MOVE.L  (B)+,D0
          BSR     WRHEXSP
          DBRA    D7,PANIC5
          LEA.L   MESS4,L
          BSR.S   WRITES
          MOVEA.L D6,B          Extract SR/PC ptr
          MOVE.W  (B)+,D0       Extract SR
          BSR     WRHEX4
          BSR.S   WRSP
          MOVE.L  (B)+,D0       Extract PC
          BSR.S   WRHEX
PANICE    BRA.S   PANICE
*
MESS1     DC.B    $0A,$0D,'Catastrophic error'
          DC.B    $0A,$0D,'Kernel start = ',0
MESS2     DC.B    $0A,$0D,'Task, Code and SSP = ',0
MESS3     DC.B    $0A,$0D,'Register dump',$0A,$0D,0
MESS4     DC.B    $0A,$0D,'SR and PC = ',0
          CNOP    0,2
```

This small subroutine is used by the panic routine to print a string, pointed at by register L (A3).

```
WRITES      MOVE.B  (L)+,D1
            BEQ.S   WRS1
            BSR     WRCHAR
            BRA.S   WRITES
WRS1        RTS
```

The next subroutine is used to print a hex number followed by a space.

```
WRHEXSP     BSR.S   WRHEX
WRSP        MOVE.B  #$20,D1
            BRA.S   WRCHAR
```

This routine writes the value held in D0 out in hexadecimal.

```
WRHEX       SWAP    D0
            BSR.S   WRHEX4
            SWAP    D0
WRHEX4      ROR.W   #8,D0
            BSR.S   WRHEX2
            ROL.W   #8,D0
WRHEX2      ROR.B   #4,D0
            BSR.S   WRHEX1
            ROL.B   #4,D0
WRHEX1      MOVE.B  D0,D1
            ANDI.B  #$0F,D1
            CMPI.B  #9,D1
            BLE.S   WRHEX0
            ADD.B   #('A'-10-'0'),D1
WRHEX0      ADD.B   #'0',D1  Drop through to WRCHAR
```

This is an assembler callable routine which must be provided. It prints the character in D1 to the primary terminal. This call is used by the panic printer, and is also used by the Tripos debugger via the standalone I/O call. The call should perform I/O without interrupts, polling until the device is ready. It should expect the device to be initialized already, possibly in the previous initialization section, and it should leave the device in such a state that the normal serial device driver is not confused. Only registers D0-D7 and A3-A4 may be corrupted.

```
WRCHAR      BSET     #0,EPCI+E_CR   Enable transmitter
            BTST     #0,EPCI+E_SR
            BEQ.S    WRCHAR         Wait for ready
            ANDI.B   #$7F,D1        Strip top bit
            MOVE.B   D1,EPCI+E_RWD  Send character
WR1         BTST     #0,EPCI+E_SR   Wait for ready
            BEQ.S    WR1
            BCLR     #0,EPCI+E_CR
            RTS
```

In the same way this routine must be provided to poll for a character and to return it in D1. Any parity bit should be stripped.

```
RDCHAR      BSET     #2,EPCI+E_CR   Enable receive
RDC1        BTST     #1,EPCI+E_SR
            BEQ.S    RDC1           Wait for char
            MOVE.B   EPCI+E_RWD,D1
            ANDI.L   #$7F,D1
            RTS
```

This next section is merely a BCPL interface to the standalone I/O routines implemented above. You should not alter them.

```
            DC.L     LIBWORD
            DC.B     7,'sawrch '
SAWRCH      MOVE.W   SR,D3          Save old status
            ORI.W    #$0700,SR      Turn interrupts off
SAWRCH0     BSR.S    WRCHAR         Write character
            CMPI.B   #$0A,D1        Was it a *N?
            BNE.S    SAWRCH1        No
            MOVE.B   #$0D,D1        Yes - send CR as well
            BSR.S    WRCHAR
SAWRCH1     MOVE.W   D3,SR          Restore interrupts
            JMP      (R)            And return
```

```
            DC.L    LIBWORD
            DC.B    7,'sardch '
SARDCH      MOVE.W  SR,D3       Save old status
            ORI.W   #$0700,SR   Turn interrupts off
            BSR.S   RDCHAR      Get char in D1
            BSR     WRCHAR      Reflect it
            CMPI.B  #$0D,D1     Return?
            BNE.S   SAWRCH1     No, exit
            MOVE.B  #$0A,D1     Yes, convert to *N
            BSR     WRCHAR      Reflect that as well
            BRA.S   SAWRCH1     Exit
```

The final section of code within the device dependent library is to restore the time from a real time clock, and to set the time into a real time clock. The code here must conform to BCPL specifications, so that only registers D0-D7 and A3-A4 may be corrupted. The return is achieved by a jump through register R.

The first routine restores the time from some hardware into the rootnode fields containing the days, minutes and ticks (1/50 second) starting from the start of 1978. If this is possible the routine returns TRUE, otherwise it returns FALSE. In the example here there is no real time clock so the routine returns FALSE.

```
            DC.L    LIBWORD
            DC.B    7,'RTIME  '
RTIME
            MOVEQ.L #0,D1
            JMP     (R)
            CNOP    0,4
```

The second routine is called to set the clock. The rootnode value is also updated at the same time. The call is made with D1 containing the number of days since 1978, D2 containing the number of minutes since the start of the day and D3 containing the number of ticks since the start of the minute. A tick occurs every 20ms. Again, in this example there is no real time clock, so the only job to be done is to update the rootnode fields.

```
          DC.L      LIBWORD
          DC.B      7,'STIME  '
* Copy into rootnode
STIME     MOVEM.L   D1/D2/D3,ROOTNODE+DAYS
          JMP       (R)
```

4.6 Device Handlers

A device handler, in Tripos, acts as an intermediary between a device driver and an application program. The device driver, as we have already noticed, accepts a standard set of commands including the important ones, read and write, which cause a read and write of a number of bytes. A device handler accepts the operating system calls and maps them to the device driver requests. We will begin the description of device handlers by looking at the simplest case, a serial line device handler.

The serial line device driver (tty-driver), you will recall, handles the read n bytes, write n bytes functions. We need to be able to support, within the operating system, a device called SER: which will handle the standard Tripos calls - open, close, read and write. The handler, therefore, is very simple. Firstly, the DevInfo structure will contain an occurrence of an entry for SER: (refer to the Chapter 3 of this manual for details of the DevInfo structure). Among the entries in the SER: assignment will be the name of the handler that is to support the device. Most of the other entries in this structure will be zero because initially there is no available task to handle the device, e.g. no loaded segment, no task identifier etc. When a program wishes to find the identifier of a handler for the device SER: (by a call to DeviceProc), Tripos attempts to load the code specified as the device handler and start up that code as a new task. The new task's identifier is then returned as the result of DeviceProc.

If, for example, the file name of the handler code for SER: is l:serial-handler then that file will be loaded into memory. The structure of the code in this segment will now be examined.

Firstly, the section of code to support the serial device will require initialization. Initialization occurs when the new task handler is sent its first packet by DeviceProc. This packet has the following arguments

```
arg1 - BSTR  The string that DeviceProc was called with
arg2 - INT   The value of the startup slot in the
             DevInfo node
arg3 - BPTR  The DevInfo node itself
```

The address of this packet is passed in A0 to the task's startup code. In the startup code for BCPL, this value is passed as the argument to start. Thus we have

```
LET start(parm.pkt) BE
$(
   LET openstring = parm.pkt!pkt.arg1
   LET startup    = parm.pkt!pkt.arg2
   LET node       = parm.pkt!pkt.arg3
```

(If you have not used BCPL before, you can find out more about this near relation of C in *BCPL for the BBC Microcomputer* by Chris Jobson and John Richards, published by Acornsoft, or in *BCPL - the language and its compiler* by Martin Richards and Colin Whitby-Strevens, published by the Cambridge University Press, which gives a full description of the language. As BCPL is one of the standard languages available from Metacomco, you can also find out more about using it from the *BCPL under Tripos* manual.)

It is useful to have the string from DeviceProc so that you could, for example, implement "SER:baudrate=9600". The initialization could, if required, parse the extra characters in the string and issue the correct set parameters call to the serial device. It is the responsibility of this handler to insert its own id into the taskid slot in the DevInfo structure. If it does not do so then a subsequent call to SER: will result in a new invocation of the SER: handler. If it does insert its own task id, then any subsequent calls will refer to this handler. In the case of this example we want it to refer to the same handler so we will insert the task id. The value in the startup slot we will arrange to refer to a structure of the following form

```
Value    Function      Description

LONG     Unit          Unit number
BSTR     DevName       BSTR to device name
```

which is similar to that used by the filing system but without the envec slot.

It is possible to set up this structure as part the DevInfo structure in the input file to SYSLINK (see Section 4.4.5 for an example).

The initialization code then finds the driver with which it is to interface.

```
LET devunit = startup!0
LET devname = startup!1
LET dev     = findnode( info.devs, devname )
```

The packets that are sent to the device drivers from the device handler are packets for which it must declare space. This handler must declare space for two packets; one a read packet and one a write packet. It will also need two flags to tell us whether the read packet is currently away at the device driver or present, unused, at the device handler. The flags are provided by setting the inpkt/outpkt pointers to 0 when the packet has been sent.

```
LET inpkt  = VEC pkt.arg3    // Driver input packet
LET outpkt = VEC pkt.arg3    // Driver output packet
LET open.for.input  = FALSE
LET open.for.output = FALSE
LET read.pkt = 0             // Application read packet
LET write.pkt = 0            // Application write packet
LET error   = FALSE

outpkt!pkt.link := notinuse
outpkt!pkt.type := devcommand.write
outpkt!pkt.id   := dev
outpkt!pkt.arg1 := devunit
```

```
inpkt!pkt.link := notinuse
inpkt!pkt.type := devcommand.read
inpkt!pkt.id   := dev
inpkt!pkt.arg1 := devunit
```

The initialization packet is then returned to indicate good or bad initialization depending on whether the device driver was found.

```
IF dev = 0 THEN          // If device was not found
$( returnpkt(parm.pkt,FALSE,error.objectinuse)
   return
$)

// Patch into the system structure

node!dev.task := taskid

// Finished with parameter packet...send back...

returnpkt( parm.pkt, TRUE )
```

The main job of the handler is to accept incoming client requests. These requests will be for an open, a close, a read or a write. The actions required will be to pass on read/write requests to the device driver. In the case we are concerned with the job is, in fact, very simple because we do not permit any extra buffering. Thus if the application requests a read then we simply pass the read request on to the device driver to read the requested number of bytes into the buffer provided. There are cases where you may want to buffer characters, in which case you would always have outstanding reads at the device. The device handler would then buffer characters, waiting for a client to read them.

The handler must now enter a loop waiting for any of a number of different requests. These will be packets sent to the device handler by an application. However, there can also be packets returning back from the device driver.

At the beginning of the main loop is a call to TaskWait(). This is a request for the next packet to arrive at this task.

```
// This is the main repeat loop waiting for an event

$( LET p = taskwait()
```

The handler now has to decide what action to take on each of the different packets which may arrive. This is normally done within a switch block, taking the packet type as argument

```
SWITCHON p!pkt.type INTO
$(
```

Firstly, there will be the cases 'open for read' or 'open for write'. These have the packet type act.findinput and act.findoutput respectively. For these two different open cases, we will find that we are passed a suitable file handle as the argument of the packet (see Section 1.3.3, "Device Handler Packet Types", of the *Tripos Programmer's Reference Manual)*. For the purposes of this handler this file handle is known as 'scb' - Stream Control Block. This file handle is already mostly filled in. The one job which we may wish to do is to set the type field of the file handle to FALSE to indicate that it is not an interactive console stream. There are a number of fields within the file handle which can be used for the storage of particular information. If we wanted a piece of information to come back, for example a pointer to some structure telling us which file handle this was, we could put that into the argument field of the file handle. If all is well, we return the packet with a TRUE completion code which indicates that we have indeed accepted the open request. This handler will only support a single serial line, so only one application task may be permitted to use that line at any one time. A flag is therefore set to indicate that the serial line is now open. If any other request to open in the same direction is received, before the previous one is closed, then it will be rejected with a FALSE result with a 'object in use' error as the secondary error code.

```
CASE act.findinput:      // Open
$( LET scb = p!pkt.arg1

   IF open.for.input THEN
   $( returnpkt(p,FALSE,error.objectinuse)
      LOOP
   $)
   open.for.input := TRUE
   scb!scb.id    := TRUE   // Interactive
   scb!scb.arg1 := act.findinput
   returnpkt(p,TRUE)
   LOOP
   $)

CASE act.findoutput:
$( LET scb = p!pkt.arg1

   IF open.for.output THEN
   $( returnpkt(p,FALSE,error.objectinuse)
      LOOP
   $)
   open.for.output := TRUE
   scb!scb.id := TRUE   // Interactive
   scb!scb.arg1 := act.findoutput
   returnpkt(p,TRUE)
   LOOP
$)
```

We now consider the close case. In close we unset the flag which says that the device was in use, and if we had perhaps allocated any memory for the open we would deallocate it here. We do not have to deallocate the file handle because that is done for us by the rest of the operating system. We also do not need to flush any buffers here because that is done prior to the call to close, again by the rest of the operating system.

```
CASE act.end:           // Close
   TEST p!pkt.argl = act.findinput THEN
      open.for.input  := FALSE
   ELSE open.for.output := FALSE
      UNLESS open.for.input | open.for.output DO
   node!dev.task := 0

   returnpkt(p,TRUE)
   LOOP
```

If we look at the calls to read and write, a packet will be sent of the format described for read and write requests as detailed in Section 1.3.3, "Device Handler Packet Types", of the *Tripos Programmer's Reference Manual*. A read request will contain the buffer pointer, the length and the argument from the file handle, which could have been an internal data structure placed there earlier in the call to open. In this case we simply hold on to the read request packet and issue a device read that is almost identical. The device request will contain the buffer pointer and length that were provided by the application. We store the application's packet to show that we are, in fact, waiting for a device request to come back. We then send the request to the device driver. Notice we do not send back the packet to the client because obviously the read has not yet completed. In a similar way we do almost identical work for the write case, sending out a write packet to the device driver and marking the fact that we are waiting for it to come back by storing the application's request packet as before.

```
CASE 'R':               // A read request

   read.pkt := p
   handle.request(devcommand.read,p,inpkt)
   inpkt := 0
   LOOP

CASE 'W':               // A write request

   write.pkt := p
   handle.request(devcommand.write,p,outpkt)
   outpkt := 0
   LOOP
```

The next two cases cover the return of the packet from the device driver. When a device driver sends back the read packet it will contain the amount of data read which will be held in the result fields of the packet. We simply transfer the result information into the result fields of the client's packet; we do not, of course, need to copy the data because we have just passed the buffer to the device driver. We can then return the client's packet and mark the fact that the device driver packet is back with us again. Again it is a very similar case for the device write return.

```
CASE devcommand.read:        // Read request returning

   inpkt := p
   handle.return(p,read.pkt)
   LOOP

CASE devcommand.write:       // Write request returning

   outpkt := p
   handle.return(p,write.pkt)
   LOOP
```

The final case in our set of statements is, therefore, merely the default case. This means that an unimplemented request has been received from the application task. The default action is to return the packet with an error indicating that the action is not known.

```
DEFAULT:

     UNLESS open.for.input | open.for.output DO
        node!dev.task := 0
     returnpkt(p,FALSE,error.actionnotknown)
```

This marks the end of the switch statements options.

```
$)
```

This handler is designed to exit when there are no streams open and no packets outstanding at the driver. When this condition occurs the following statement will terminate.

```
$) REPEATWHILE open.for.input | open.for.output |
                outpkt = 0 | inpkt = 0
```

At this point any closedown action, for example deallocation of buffers, would take place. However, in this case no action is required and the task simply exits.

```
$)
```

The last two routines in this task handle the sending of packets to the driver/application with the appropriate arguments/results.

```
AND handle.request(command, rp, tp ) BE
$(
   LET buff = rp!pkt.arg2
   LET len  = rp!pkt.arg3

   tp!pkt.arg2 := buff
   tp!pkt.arg3 := len
   qpkt( tp )
$)

AND handle.return(rp, p ) BE
$(
   returnpkt(p, rp!pkt.res1, rp!pkt.res2 )
$)
```

4.7 Porting Tripos

The porting process involves several distinct stages, starting with the terminal driver and finishing with a complete bootstrappable system.

The stages are as follows:

1. Terminal driver for a non-disk based CLI
2. Clock driver
3. Disk driver
4. Format and initialise the system disk
5. Download system commands and information files
6. Downloading and installing the Tripos system image

WARNING: These six stages may take several weeks to complete!

---

1. Terminal Driver for a Non-Disk Based CLI

You need to write the terminal driver first so that you can test user communication. As the filing system and restart segments are not needed at this stage, your syslink input file should look something like this:

```
absmin          #X0000;
absmax          #X032F;
storemin        #X4000;
storemax        #X8FFF;
memorysize         36;
rootnode          512;
tcbsize            29;
mcaddrinc           4;
```

```
seg syslib  klib.obj,jacket.obj,mlib.obj,dlib.obj,
            blib.obj,extras.obj,io.obj;
seg debug   taskint.obj,debug.obj,debug-disasm.obj;
seg cohand  taskint.obj,cohand.obj;
seg cli     taskint.obj,cli.obj,cli-init.obj;

tasktab 20;

*task   1 pri 1000 stack 400 segs syslib,cli;
 task   2 pri 2000 stack 300 segs syslib,debug;

devtab 20;

dev -2 driver tty-driver.bin;

info (
      "68000",
      ((0,
          0, 0, 0, 0, 300, 2999,
             (1, "serial"), cohand, 0, "AUX"),
          0, 0, 0, 0, 300, 3000,
             (0, "serial"), cohand, 0, "CON"),
      ((0,
          "cli",      1, cli),
          "syslib",   1, syslib),
      (0,
         "serial",   1, -2),
      0
     )
```

This definition shows a system with a terminal driver as driver -2. This driver is capable of handling two units known logically as CON: and AUX:. The driver is known by the name of "serial" and it is by this name that the console handler is able to find it. When the initial CLI starts up it attempts to assign the logical name SYS: to the root directory of the first handler on the handler chain. In this case it, for instance, would attempt to assign SYS: to CON: (the chain is CON: AUX:). This would fail, however, as CON: is not a directory-structured device. The CLI then starts up in a non-disk-based mode whereby it attempts to find programs to run from the resident segment list.

You may add any command to the resident segment as follows:

In the segment declarations:

```
seg status status.obj;
seg echo   echo.obj;
```

In the resident segment list:

```
((((0,
   "cli",      1, cli),
   "syslib",   1, syslib),
   "status",   1, status),
   "echo",     1, echo),
```

The commands STATUS and ECHO are added to the resident segment list. (This means that you can now give these two commands, if you wish.) Note that according to this structure a non-disk-based CLI could find and attempt to run the commands 'SYSLIB' and 'CLI'. However, this should not be attempted, since those segments are not commands but are set up only so that the system can find them when starting up a new task/CLI.

Note: You should not use driver -1 at this stage as many parts of the system assume this to be the clock driver.

2. Clock Driver

Now you can proceed to make the clock driver work. You do this because the filing system, which is the next stage, uses the clock driver to time its periodic disk checking. The device declarations part of the syslink input will now look like this:

In the device table initialisation:

```
dev -1 driver clock-driver.bin;
```

In the device list:

```
((0,
   "clock",    1, -1),
   "serial",   1, -2),
```

3) Disk Driver

The disk driver is the next object in the process. This driver should be able to read, write, format and acquire status information from a disk. It should also be able to handle the 'motor off' command but this may possibly be a null action. To test the disk driver the filing system needs to be loaded by syslink.

This is done as follows:

In the segment declarations:

```
seg fihand    taskint.obj,access.obj,bitmap.obj,
                bufalloc.obj,disc.obj,exinfo.obj,
                main.obj,support.obj,work.obj,
                moveb.obj,init.obj,state.obj;
seg restart   taskint.obj,restart.obj;
```

In the device table initialisation:

```
dev -3 driver disc-driver.bin;
```

In the info structure initialisation:

```
(
 "68000",
 (((0,
        0, 0, 0, 0, 300, 2999,
           (1, "serial"), cohand, 0, "AUX"),
        0, 0, 0, 0, 300, 3000,
           (0, "serial"), cohand, 0, "CON"),
        0, 0, 0, 0, 210, 2500,
           (0, "disc", (11,256,0,2,1,4,4,0,0,0,79,5)),
              fihand, 0, "DF0"),
 (((0,
      "restart", 1, restart),
```

```
        "cli",      1, cli),
        "syslib",   1, syslib),
    (((0,
          "timer",    1, -1),
          "serial",   1, -2),
          "disc",     2, -3),
    0
   )
```

This defines the disk-driver as device -3 and names it "disc". The assignments list is initialised such that there is a disk known as "DF0:" which has 80 tracks (0-79), double sided, 8 * 512 bytes per sector. There is also one reserved block which is used by the bootstrap. This is the standard configuration of a 5 1/4" floppy disk.

4. Format and Initialise the System Disk

To format an unformatted disk, you need to have the FORMAT command resident. You can make this resident as described above by declaring a segment and putting it into the resident segment list. (See 1. Terminal Driver for further details.) The command line

```
> FORMAT DRIVE df0: NAME "System disk"
```

then formats the disk in drive "df0" and initialises it with name "System disk".

5. Download System Commands and Information Files

Now download commands from the development system using a downloading tool. For example, you can use the program "Kermit" which is provided and which will perform this job well. Make Kermit resident as shown above, and then type

```
> KERMIT aux:
```

to run Kermit in local mode.

Since the CLI is non-disk based it does not initially have a current directory to the Kermit command:

```
Kermit-68K (Local)> setdir df0:
```

is needed to set to that disk.

Set Kermit to image mode transfers with the command:

```
Kermit-68K (Local)> set image on
```

Then connect to the development system with the 'c' command.

You may now run the development system's Kermit to download all the command directory files. Use the Tripos Kermit command 'r' to receive files sent from the development system.

Once you have downloaded a command directory file, place it in a directory named 'c' on the system disk (sys:c). When the CLI starts up it will assign the command directory C: to the directory sys:c.

Place the vdu information file in a directory devs/vdu.

6. Downloading and Installing the Tripos System Image

The Tripos system image file should now be downloaded into the directory sys:l.

The installation process is often system dependent but it always consists of some method of informing a bootstrap program of exactly where on disk to start loading a system image file.

The system image file is in the form described under "Object File Format" and will only contain "hunk_abs" sections terminated by a "hunk_end".

One method of installation that is often used is to plant the key number of the image to be loaded into a table of keys and then to save this table in the first block on the disk. This table of keys is then indexed by a version

letter so that multiple Tripos images may be installed at the same time on the same disk.

Now you should write the bootstrap program. This is a small program typically residing in ROM which requests a drive name and image version letter from the user, loads a system image into RAM, and jumps to the system at the address stored in location ROOTNODE + KSTART.