MDEX for the CORTEX

USER MANUAL

Microprocessor Engineering Limited
21, Hanley Road, Shirley, Southampton, SO1 5AP
0703-775482

MDEX for the Cortex - User Manual

Rev 1.3

Microprocessor Engineering Limited
21, Hanley Road, Shirley, Southampton, SO1 5AF
0703-775482

MDEX for the Cortex - User Manual

CONTENTS

Microprocessor Engineering Limited
21, Hanley Road, Shirley, Southampton, SO1 5AP
0703-775482

January 84

1. Introduction

The Marinchip Disc Executive (MDEX) is a disc operating system for small computers based around any of the 99xx(x) range of processors. It has been running in commercial service since before 1978. It is supplied with many useful tools and utilities, and many purchasers who have bought the complete package will no software other than that which comes with MDEX. Users who have bought MDEX in its unbundled form will find alternate procedures documented here as required.

MicroProcessor Engineering has now converted MDEX to run on the Powertran Cortex. Because the system originally ran with 8" disc drives, a 24 x 80 VDU, a parallel printer, and an S-100 crate (Marinchip were the first manufacturer to run a true 16-bit system on the S-100 bus), there are some differences and changes to the system that is described in the Marinchip documentation.

The aim of this section is to introduce you to the system, show you how to get started, and to persuade you not to panic when all seems to be lost. It isn't lost - you just don't know where it is, or where you went afterwards.

READ ALL OF ME FIRST

2. Starting out

Operating systems are here to make your life easy - they are tools, and take a little bit of getting used to. The reason for having an operating system is that having to remember how the hardware works every time you want to write a program is a bore. So we write a set of programs which are loaded into the computer before any other programs are run. This first set contains programs and subroutines which look after and control the hardware. As well as meaning that you don't have to remember how the hardware works, it now becomes easy to change the hardware, as the only software that has to be changed is in the operating system, not in your carefully crafted programs. If you write a program to run on the Cortex using MDEX it will run unchanged on any Marinchip system.

Obviously there will be changes for things the operating system cannot control, such as the type of terminal used, but these changes are usually very easy.

3.1. Loading the operating system

Before any of your programs can be run we must first load the operating system itself. Rather than doing this from tape, we can do this from a disc. Throughout this manual, we assume that you are a using a Cortex with two 5" disc drives. Operation for other versions is very similar.

You should have at least two discs with the the MDEX package. One is called the "boot disc", and the second is called the "system disc". There will also be one or two other discs, but they will be labelled System Generation Kit or SGK - you will not need them until later, if at all.

Go out and buy a box of discs for double density use. If you insist on using the system before you have copied the discs we sold you - on your head be it - don't say we didn't warn you. The only discs we stake our life on are made by DYSAN, they are worth the extra money. Cheap discs will at best make the disc heads dirty, and at worst ruin discs and drives. After loading the operating system you should copy the discs - read on for instructions.

The reason for this heart-stopping warning is that if you ruin the the discs we sent you BEFORE you copy them, you won't be able to play until after you have admitted this to us, and obtained a new set of discs; BUT if you have copied the discs, all you have to do is make another set, and gaily start again.

WHEN ALL ELSE FAILS - READ THE INSTRUCTIONS

You will have realised that the most dangerous part in the whole exercise is getting as far as copying the discs. This part even causes professionals to twitch erratically. Terror subsides in the face of knowledge, and the first important fact is that disc drives will not write onto the disc if the disc is "write protected". On 5" discs there is a notch to the right of the label on the right hand edge of the disc jacket. If this notch is uncovered, take a little sticky bit out of your new box of discs and cover up the notch. The disc is now write protected. Do this with the other discs. On 8" discs the notch is on the bottom edge, to the right - it must be uncovered to write protect the disc. You didn't think that computing would really be any different, did you ?

The only way you can ruin the disc now is to turn the computer on or off with the disc in the machine, or to stick your fingers on the magnetic part of the disc and make it dirty.

The Basic supplied in EPROM with the Cortex copies itself into RAM at power up, and then turns the EPROMs off. One of

the Basic commands is 'BOOT'. This command reads a little program from the disc and runs it. As MDEX uses this same part of the disc for something else, you will have to tke the first disc out, and replace it with another disc that actually has MDEX on it, and the little program loaded by the 'BOOT' command will then load MDEX itself into the Cortex.

So, place the disc labelled 'boot disc' into the left-hand disc drive (label upwards and nearest you). Type 'BOOT', the disc drive light will come on, and the go out after a second or two. If the light does not go out - have you closed the disc drive door ? - if not close it. If the light still does not go out, open the door, take the disc out, and press the reset button on the back of the Cortex. The most likely reason for failure is that the links on the board are not set for booting a single density 5" disc.

Eventually, the disc light will go out a few seconds after you typed 'BOOT'. The little program that actually loads MDEX has been fetched from the disc, and is running. It is waiting for you to change discs.

So, open the disc drive door, take out the 'boot disc', replace it with the 'system disc', and shut the door again. Now place a blank disc in the right hand drive, and shut its door. Press any key, lights will light, and if all goes well there will be a message on the screen. If there isn't, check that you did have the 'system disc' in the left hand drive.

If you got the sign on message - MDEX is running. If you can't get anything to happen read this section again, try again and if all else fails contact MPE.

2.2. Copying discs

The boot disc has been recorded in single density format, and the system disc is in double density format (you can get twice as much on it). To make the copies you must format your new discs, one in single density, one in double density. On 8" systems both discs are in single density.

Leave the system disc in the left hand drive and type FORMAT to call up the disc formatting program. There is a section of the manual about the disc formatter later on. READ it. Format at least two new discs, one in single density for the boot disc, and one in double density for the system disc.

The formatted discs are now usable but first must have a new directory written on them. Without this, the operating system will not know how to use the disc. The command to prepare a new directory is PREP, but first read the section on PREP in the Marinchip documentation. In practice the boot disc does not need the PREP procedure. To prep a disc in drive 2 for double density use type PREP 2/,DD<cr> and to prep a disc for double sided double density use type PREP 2/,DS,DD<cr>. For single sided single density just type PREP 2/<cr>.

You now have two formatted and prepared discs. Copying

information from one to another is easy. There are two methods of doing this. The one described here is only applicable if you have bought the full version of MDEX, if you have not you must use the COPY utility.

2.2.1. Copying discs using DU

Type DU to get the disc utility. As the system disc is already in the left-hand drive and is in double density format, place a double density disc in the right
and now type CD 1 2. This tells the disc utility to copy all the data on disc 1 onto disc 2. Remove both discs, and label the new one using a sticky label from the box, and a felt-tip pen. DO NOT use pencil or biro. Insert the boot disc in the left hand drive and a single density disc in the right hand drive and again type CD 1 2. If you are really cautious you will now make another set of copies, as the learning period is the one in which are most likely to need to use the issue discs or their descendants.

Now put the issue discs (the ones we sent you) away in a safe place and do not use them for anything except getting yourself out of a hole. If you value the work you do on this machine and will get upset if your discs are corrupted, do buy quality discs - they last longer and don't clag up the heads on the disc drives. It is not often that we recommend products, but if you can afford them, do use Dysan discs for those working discs that will get a lot of use.

2.2.2. Copying discs using COPY

The COPY utility is described
to copy discs from one drive to another place a formatted and PREPped disc in drive 2, the disc you want to copy from in drive 1 and type COPY 2/=1/*.*<cr> to perform the copy. Answer Y<cr> in response to the question 'Auto-create enable ?', and all the files on drive 1 will be copied onto drive 2.

2.3. Seeing what you've got.

You can find out what is on the discs by typing CAT (for catalog).

3. Operating differences from the normal Marinchip MDEX

All the differences between MDEX for the Cortex and for other machines stem from the Cortex having a 40-column screen, and all the other systems have 64 or 80 column screens. Some programs and utilities become less easy to use because of this, and have been re-written. The disc directory utility DIR assumes an 80-column screen, so we have provided you with CAT. The disc formatter, FORMAT, is unique to the Cortex because it is hardware dependent.

Creating a working disc

We have found that the easiest way to work is with all the tools you need on one disc, and the job itself on another. After you have FORMATted and PREPped a working onto it (see Marinchip documentation for TCOPY and BCOPY) the following files: SHELL$.OBJ, TCOPY, and CAT. You will soon add to this list a few utilities that you use all the time when copying new files onto your job discs, and consequently you have to remove the system disc.

5. Disc Formatter - FORMAT

FORMAT was written as a general purpose disc formatting tool, and can be used to generate many different formats. Type FORMAT to load the program (the cowardly then remove all discs except the one to be formatted). FORMAT then asks you a set of questions about the disc format required.

Cortex internal drives are 5". 40 track (80 track on 'D' models), 16 sectors/track, and 128 bytes/sector (256 in double density). 'D' models have double-sided drives. In order to format double-density discs answer 'd' to the density question, and select 256 bytes/sector. Again, please note that FORMAT is a general purpose program which will format discs to run on many different machines. For use with MDEX, discs must be PREPped (prepared) to write a blank directory onto the disc.

The larger 8" drives differ in having 77 tracks of 26 sectors.

If you need special formats for any reason contact us for a custom version of FORMAT.

6. CAT disc directory program

As the disc directory command DIR assumes an 80-column screen we have written a new program CAT to provide a catalogue of what is on the disc. Ensure that CAT is on the disc in drive 1, type CAT, and then tell CAT which drive you need a directory listing for.

7. COPY file transfer utility

This program is not a Marinchip product but has become part of the system. It is a general purpose file transfer program, and its great virtue under MDEX is that it will create its own files. A second virtue is that it will accept an asterix '*' as a wild card, so that the command - copy 2/*.*=1/*.* - will copy all the files on drive 1 onto drive 2, and will automatically create the files if you answer y to the question 'Auto-create enable (Y/N) ?'.

There are several forms of the copy command:-
COPY 2/=1/*.* will copy all the files on drive 1 onto drive 2
COPY 2/*.*=1/*.* is like the previous example

COPY 2/=1/FRED.ASM will only copy the file FRED.ASM from drive 1 onto drive 2
COPY 2/=1/FRED.* will copy all files called FRED with any extension to the filename
COPY 2/=1/*.ASM will copy all files with the extension ASM

8. Boot files

The secondary boot program (on track 0 of the boot disc) has been rewritten so that it will boot from single or double density discs. The source of this program is provided as part of the System Generation Kit, and the linked output is provided as SBOOT.5S for single sided, and SBOOT.5D for double sided disc drives. There are several other files named BOOT.ijkl, where i and j refer to the size and sides of drive 1, and k and l to the size and sides of drive 2. To generate an operating system with these new characteristics, copy the desired file into BOOT$.SAV (on a fresh disc!), and re-boot using the new disc.

9. Terminal control codes

On the Marinchip development system at MPE, the terminal is a Televideo 920C. When we moved MDEX to the Cortex, we used the same control codes to perform certain functions on the Cortex screen. Thus an application program should treat the Cortex screen as if it is a TVI-920C terminal.

Not all the codes are implemented, but the ones described below are implemented. Customers who have also bought the System Generation Kit (SGK), will be able to add extra codes as required.

function	character sequence	code(s)
bell	control-G	07
cursor left	control-H/backspace	08
cursor down	control-J/line-feed	0A
cursor up	control-K	0B
cursor right	control-L	0C
home cursor	control-^ or ^	1E
clear screen	ESC + or ESC Z	1B,2B or 1B,5A
	or control-Z	1A
clear to end line	ESC T	1B,54
clear to end screen	ESC Y	1B,59
gotoxy	ESC = y-pos x-pos	1B,3D,y+20,x+20

10. Using the serial port for a terminal

The operating system as delivered uses the Cortex screen and keyboard as the terminal. When the operating system signs on the serial port is initialised to 9600 baud, and the cassette port is initialised to 1200 baud. By changing various flags within the operating system, your programs can arrange to switch between using the keyboard and the serial port to receive data, and between the Cortex screen and the serial port to display data.

These flags are in what is called the 'unit table' for the terminal. See below for a description of the unit tables.

Modifying device drivers

If you need to perform permanent changes to the operating system we recommend that you use the System Generation Kit to modify the device drivers (the bits of the operating system that control the hardware). For small changes or in particular for changing the behaviour of the screen and keyboard, the changes can be made from an application program, or by using the Debug Monitor.

Each device - terminal, two disc drives, and printer - is described to the operating system by a unit table, and within limits these can be modified by a user.

The easiest way to experiment is to use the Debug Monitor to examine and/or alter the contents of these tables. Type DEBUG<cr> to load the debug monitor, and when you have finished use GC008 to reinitialise the MDEX operating system. Address C008 is the cold start entry point of this issue of MDEX. To check it look at the first sector of BOOT$.SAV using DU. In this sector are three words which define the file's wokspace, entry point, and length.

Sections of code are given from each of the drivers to show how to address them. Each item uses two bytes at an even memory address. False is indicated by a 0 in the location, and true by non-zero.

The address of the start of the table can be found at fixed locations - although the table addresses may (and will) change as MDEX is modified, they will always be found at the fixed locations in low memory. These pointers are only available on MPE's versions of MDEX for the Cortex - they are not available on other implementations.

hexadecimal address	points to table name	controls device type
0FE	ut$con	terminal/screen
0FC	ut$fd1	disc drive 1
0FA	ut$fd2	" " 2
0F8	ut$prt	printer

Your discs should be backed up before you start experimenting with these tables, there is no protection and if you get it wrong the results may be unpredictable - sometimes rather unpleasant. In particular do not change the bytes/sector of the discs from 128 - unblocking routines are not fitted.

Disc unit table

The following cells occur for all units

```
*
            dorg    0
master      equ     $
*
*
dtmstr      bss     2               link to first unit on controller
dtselb      bss     2               select bits for this unit
dtcurtrk    bss     2               current track position
dtprect     bss     2               precession table
dtmfm       bss     2               nonzero if last I/O was d/density
dt2hd       bss     2               nonzero if d/sided disc mounted
dtbytsec    bss     2               bytes/sector (single density)
dtsectrk    bss     2               sectors/track
dttrksrf    bss     2               tracks/cylinder
dtsize8     bss     2               non-zero for 8" drives
*
lenslave    equ     $-master
*
*           The following cells exist only in the control table for the
*           first unit on the controller.
*
dtdeva      bss     2               device address
dtbuff      bss     2               DMA buffer address
dtsvbuf     bss     2               save sector buffer
dtdtrk      bss     2               desired track number
dtdhead     bss     2               desired head number
dtdsec      bss     2               desired sector number
dtmema      bss     2               memory buffer address
dtmeml      bss     2               memory buffer length in bytes
dtcfunc     bss     2               current function save
dtretrvc    bss     2               retry countdown cell
dtodtrk     bss     2               original desired track
dtodhead    bss     2               original desired head
dtodsec     bss     2               original desired sector
dtomema     bss     2               original memory start address
dtomeml     bss     2               original memory length
dtdmam      bss     2               DMA mode
dtst0       bss     2               fdc status
dtemp1      bss     2               temporary cell 1
dtemp2      bss     2               temporary cell 2
dtrbw       bss     2               read before write done flag
dtrvrtry    bss     2               read verify retry count
*
lenmaster   equ     $-master
            rorg
*
*           unit definition table for terminal
*
            dorg    0
conbeg      equ     $
*
dtmaster    bss     2               link to master console
dtserip     bss     2               1 for serial i/p, 0 for parallel
dtserop     bss     2               1  "    "    o/p, 0  "     "
dtcruser    bss     2               cru address of serial 9902
dtbaudser   bss     2               timer value for serial tx/rx
*
maxcol      bss     2               characters/row
```

```
a.row     bss     2                 rows/screen
a.pos     bss     2                 max chars/screen (maxcol*maxrow)
cur.pos   bss     2                 current cursor position
ursor.on  bss     2                 cursor required flag; 1=on
avecursor bss     2                 temp storage
esccount  bss     2                 nth character in escape sequence
escchar   bss     2                 second char in escape sequence
ewrow     bss     2                 new row from esc = sequence
ewcol     bss     2                 new column from esc = sequence
*
onlen     equ     *-conbeg          length of definition table
          rorg
*
          Printer control table

          dorg    0
*
rdvad     bss     2                 printer device address
prcrdly   bss     2                 carriage return delay time
prautolf  bss     2                 cr does lf if nonzero
rvpos     bss     2                 vertical position in page body
rffdly    bss     2                 form feed delay time
prbody    bss     2                 page body length
rtop      bss     2                 lines at top of page
rminl     bss     2                 minimum line length
prhpos    bss     2                 horizontal carriage position
rlfdly    bss     2                 line feed delay
rffsim    bss     2                 simulate form feed if nonzero
prbot     bss     2                 lines in bottom margin
*
          rorg
```

12. Double-sided disc drives

If you have a machine with 8" double-sided discs, ensure that the Cortex can use the signal from the drive ("two-sided") that indicates whether or not the disc in the drive is a double-sided disc. Without this signal, MDEX can't distinguish between single and double-sided discs. This signal is not provided on the PCB.

Machines supplied by MPE with double-sided drives will have been modified and supplied with MDEX configured to use both sides of the disc. If you have to modify the Cortex, here are the instructions:-

1) Connect a link between the two-sided signal line on the floppy-disc connector to IC 63 pin3 (ETI numbering). This is a 74LS251 used as an input port. The two-sided signal is usually to be found on pin 10 of the 50-pin connector for 8" drives, and is not defined for 5" drives.
2) Install a 150R pull-up resistor between IC 63 pin 3 and +5v (IC 63 pin 16). This provides for termination of the signal from the drive.
3) If you have an early version of MDEX for the Cortex, and are converting from 40 track drives, MDEX may not support double-sided drives. In this case use the System Generatio Kit (SGK) to add the required code to the disc drivers, or

contact MPE to obtain the update.

13. Common disc problems

Some people have had trouble in getting MDEX running. This is usually caused by hardware faults that can easily be corrected.

Ensure that the side select line is connected to the 5" disc connector. On some issues of the Cortex main PCB, the side select line exists on pin 14 of the 8" (50 way) disc drive connector, but not on pin 32 of the 5" (34 way) connector. Double sided 5" systems will load the boot disc, but not the system disc if this link is not made.

In the phase lock loop section of the disc controller are three resistors which control the time period of a monostable used in the disc read circuitry. These are labelled R68,69,70 on our circuit diagram, page A12. The original values were 4k7, 10k, and 18k. More recent machines are fitted with 2k7, 5k6, and 12k. The 5k6 value is a compromise between the values required for 5" double density and 8" single density use. For 5" discs this value can often be reduced. Several users have reported optimum results using 3k9 or 3k6 in this position.

The disc drives must be correctly configured. There are several banks of switches on most drives. For 5" drives ensure that the drives are selected using DS0 and DS1 as drives 1 and 2 respectively. Note that on any one drive only one of DS0-3 must be closed. Only one of the switches labelled HS and HM should be closed. These switches are used to select whether the head is loaded only when the drive is selected (noisy), or whenever the motor is turned on (leaves the heads down longer). In general set HM and have a quiet life. Usually there is a switch called MX which should be left open. These switch settings are not guaranteed for all drives, but are the settings we use first when we get a type of drive we have not used before.

We have had no trouble with the TEAC drives, the 55F type being very economical as a high storage capacity drive.

Use with 8" double density is often difficult.

14. Miscellany

This section contains odds and sods of information which we find useful, but won't fit easily into the rest of the documentation without a lot of work.

The file SHELL$.OBJ contains a collection of programs like DIR and CREATE, and so must be present in drive 1. Unlike other disc-resident commands, you cannot use these commands by putting SHELL$.OBJ on drive 2, and then typing 2/CREATE etc.

File copying is quicker with BCOPY than with TCOPY, and

easiest with COPY.

The assembler error messages are somewhat terse, and can be the result of previous errors, rather than an error on the current line. In particular, failure to use EVEN after a TEXT statement can leave the location counter at an odd address. This will cause an error in the next executable instruction. If the assembler cannot open a 'copy' file, you may well get an error message on the next line.

New discs do not have to be logged in, but do not change discs while a file is being used by a program. The operating system will detect the disc characteristics (single/double density, single/double sided) when the directory is read at track 0 sector 1). Automatic single/double sided disc selection can only be performed when using drives with a two-sided signal connected to a Cortex/PP95 modified as described previously. When using the disc utility DU, you must read track 0 sector 1 if you change the type of disc in a drive.

15. Technicalities and legalities

MDEX is issued on a single machine licence. This means that you are only supposed to run it on one machine. Of course you can make as many back-up copies as you like, but you are not allowed to pass it on. Because some people may need to run MDEX on more than one machine, further copies may be purchased at a reduced rate. Contact us for these prices.

The greater part of the purchase price of MDEX (and all the other Marinchip software) goes to people who put in a lot of time to generate and support this software. Marinchip are good people and deserve their money.

We like to think that we deal fairly with you, our customers. If you agree that we are dealing fairly with you, I hope you will be fair to us by respecting the copyrights on our software, and by not making copies for others.

MDEX USER GUIDE - 1

by John Walker

Marinchip Systems

Mill Valley, CA 94941

Marinchip 9900 Disc Executive User Guide

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

1. Introduction

The Marinchip 9900 Disc Executive is an operating system for the Marinchip 9900 computer system. It provides a comprehensive set of user services, facility allocation and resource management features, and requests available to programs running under its control. Key features of the operating system are:

- Named files on disc. All disc I/O is file relative. The system performs all disc space allocation and detects attempts to write or read outside file boundaries.

- The system contains all I/O drivers. All system peripherals are handled as files within the file system. Programs running under the system can use disc files or hardware peripherals such as printers without modification. All programs running under the system are independent of the hardware.

- The system allocates memory automatically to programs. Programs running under the system can automatically adapt to the amount of memory available without regeneration.

- The system provides powerful local editing on the terminal used to run the system. Features include backspace to correct errors, word delete, and a key that retypes the current input line as edited. Programs need not contain code for these functions. Also, the user sees a consistent terminal interface across all programs.

- The system calls are upward compatible with the Marinchip Network Operating System. Any program that runs under the Disc Executive will also run under the Network Operating System. This protects the user's programming investment when moving to the more advanced system.

- The system command set can be extended simply by writing user programs. When the system encounters an unknown command, it simply loads a user program with that name (if one exists). This allows a custom system to be built with no modifications to the Executive itself.

2. Using the system from a terminal

This chapter describes the Executive system as seen by the user at a terminal. This is a complete description of the system for all

users except those coding Assembler programs which call the system. Assembly language interface information will be found in the chapter "Using the system from a program" later in this manual.

2.1. Loading the system

Depending upon the system configuration, the system may be either automatically loaded when power is applied to the machine, or may have to be loaded from the Marinchip 9900 Debug Monitor. If the Debug Monitor is configured, the system is loaded by entering the command:

```
BOOT
```

When the system has been successfully loaded, the system sign on message:

```
Marinchip Disc Executive (Ver. x.x)
```

will appear, and a period will be typed on the next line. This period is the "command prompt", and will be typed whenever the Executive is ready for another command. When the period appears, any of the system commands described below may be entered, or the name of any executable file in the file system may be typed. If a file name is typed, it will be loaded and executed under the control of the system.

2.2. The file system

2.2.1. Disc files

The Marinchip Disc Executive provides named files on disc, and files to support all hardware peripherals. File names are twelve or fewer characters chosen from the upper case ASCII letters, the numbers, and the special characters:

```
- $ .
```

Lower case letters, if used, will be treated identically to upper case: there is no difference in the names "Fname" and "FNAME". One disc unit in the system configuration is referred to as the "system disc". This unit, always called unit 1, is the unit from

which the Executive is loaded, and is the default unit on all file references. References to file names on system commands are always of the form:

```
        ⟨unit⟩/⟨filename⟩
 or:    ⟨filename⟩
```

In the first case, the file with name ⟨filename⟩ on disc ⟨unit⟩ is selected. In the second case, the system disc unit, 1, is assumed. Hence the two specifications:

GNORK and 1/GNORK

are equivalent. Files with the same ⟨filename⟩ may exist on any number of separate disc units simultaneously.

2.2.2. Device files

All system peripherals such as the console and printers are included in the file system. These "Device files" are given special names within the Disc Executive. All Device Files are assumed to exist on the system disc, so that no ⟨unit⟩ specification need be given. The device files present in a system depend upon the system configuration. Only the console device file and the parameter string device file are always present. The standard names assigned to device files are as follows:

CONS.DEV	Console
DISC$x.DEV	Disc (unformatted access)
PARAM.DEV	Program parameter string
PRINT.DEV	Hard copy printer

The only restrictions in use of device files stem from their hardware limitations: it is meaningless to input from the printer, or to rewind the console. Attempts at such levity will normally be ignored by the system.

2.2.3. Common file nomenclature

Since disc files and device files can be used for the most part interchangably, throughout the rest of this manual they will be referred to by the generic term ⟨file⟩. Where the manual says a ⟨file⟩ should be named, either a device file or a disc file may be used, and either the ⟨unit⟩/⟨filename⟩ or ⟨filename⟩ form of the name may be used.

2.3. User console commands

The Disc Executive contains a minimum set of commands available to the user from the console. Most of the "commands" typed by the user for such functions as editing a program, etc., are actually names of files containing programs that perform those functions. The following commands are actually performed by the system and, as such, may not be redefined by the user.

2.3.1. JUMP - Activate program in memory

JUMP ⟨workspace⟩,⟨entry⟩

The JUMP command will transfer control to a program in memory at the address ⟨entry⟩, with initial register workspace at ⟨workspace⟩. Both ⟨workspace⟩ and ⟨entry⟩ are assumed to be hexadecimal numbers; no leading zero is required before them. This command is normally only used by machine language programmers who are patching programs in memory.

2.4. Console support

The Executive contains an extensive handler for the console through which the user interacts with the system. Since the user spends so much time using the console, the system goes to great pains to make the interaction as pleasant as possible.

2.4.1. Console input

The user may type input on the console whenever a prompt appears from the system or a program has requested input. Once the first character of input is typed by the user, all output will be held until the line is either entered by pressing the RETURN key or struck out. Several special functions are provided by control keys while input is being entered.

2.4.1.1. Line delete

The entire line of input entered so far by the user may be deleted by pressing the Control X key (ASCII code CAN). This will echo ^X on the console, throw away the line typed in so far, and retype

the prompt for the line (if any). The user may then re-enter the input from the start. If the system is configured to use a CRT terminal (VDU) as the system console, the input line will simply be erased when Control X is typed.

2.4.1.2. Character delete

The last character typed may be deleted by pressing the Backspace (Control H) key. If the console is a CRT device, the character will be rubbed out on the display and the cursor will back up. Any number of characters may be rubbed out by successive depressions of the backspace key. If all characters on the line have been rubbed out, the backspace will be ignored.

2.4.1.3. Word delete

The last word typed may be rubbed out by pressing the Control W key. This will delete characters starting from the end of the line and working towards the start until an alphanumeric character is encountered. Then alphanumeric characters will be deleted until a non-alphanumeric is found. This will have the result of rubbing out the last word entered. If the console is a CRT display, the word will physically disappear from the screen and the cursor will back up over it.

2.4.1.4. Retype input line

All input line editing with the above special keys is very easy to understand and use if the console is a CRT display. If the console is a hard copy terminal, however, overtyping many characters may make it very hard to ascertain just what is about to be sent as input. Pressing the Control R key will retype any prompt for the line, then type the current input line as it stands. The carriage will be left at the end of the input line so that further corrections may be made, if required. This key may be used at any time when entering input.

2.4.1.5. Expansion of control characters

ASCII control characters that do not have special editing functions documented above will be expanded when echoed to an up-arrow (^) followed by the letter which one presses along with

the Control key to generate the code. This feature allows easy editing of input containing control characters without the confusion of trying to edit characters that aren't visible.

2.4.1.6. Escape input

Any ASCII character can be entered as input by preceding it with the Escape key. The Escape will not be echoed, and the character following it will be echoed directly to the terminal and placed in the input buffer. This allows carriage return or any of the local editing characters to be treated as normal characters and input to a program. Note that to input the Escape character itself two Escapes must be typed, as the first forces the second as a normal character.

2.4.2. Console output

Output sent to the console by programs will simply be typed as sent, except that line feeds will be inserted automatically following carriage return characters, and delay characters will be automatically inserted to accomodate the carriage return, line feed, and form feed delay requirements of the console device. Note that control characters sent to the console by programs will not be expanded into the "up-arrow" form. This allows programs to freely send control characters that perform special functions on the console device.

2.4.2.1. Output pause

Pressing the Control S key while the output is being sent to the system console will cause the system to pause at the end of the next output line. The system will send no more output to the console until Control S is typed again. Thus, Control S may be used as a "push-push" switch to halt and resume output.

2.4.3. Console interrupt

An executing program may be interrupted by pressing the Control C key during console input or output. The input or output will be aborted. If the program has requested the console interrupt, it will be diverted to its interrupt point so that the interrupt may be serviced. If the program does not service the console

interrupt, it will be terminated and its workspace registers will be dumped.

2.5. Printer support

The system contains a special driver to format data sent to the device file PRINT.DEV. This driver performs page formatting and other control functions suited to the printer connected to the system.

2.5.1. Page formatting

When the system is generated, the physical properties of the printer are selected. The system will automatically format the page into a top margin, a body portion, and a bottom margin. Output sent to PRINT.DEV will appear in the body portion. The top and bottom margins will be left blank, and prevent user data from being printed across the page perforations.

2.5.2. Printer pause character

When the ASCII ENQ character (code 5) is sent to PRINT.DEV, the printer will enter a pause state. The printer will immediately stop, and the console will sound continuous bell signals. Pressing the space bar on the console will silence the alarm. When the RETURN key is pressed on the console, the printer will resume operation. This feature allows a program to cause the printer to stop at points during the output, permitting the user to change paper or print elements, or manually insert text in the output.

2.5.3. Console output to printer

If the ENQ character (Control E) is typed on the console, the status of echoing console input and output to the printer is reversed. Initially, console input and output are not sent to the printer. Pressing Control E will cause all console input (except direct mode input) to be echoed to the printer, and all console output also to be echoed. Pressing Control E again will turn this mode back off. Control E may be used either during console input or output, and has no effect other than setting the printer mode (it will not go into the input buffer if entered as part of

console input). Console data are echoed to the printer on a line by line basis, not character by character. As a result, only "clean" data (after all local editing, etc.) are printed, so this mode is ideal for preparing samples to explain how to use the system.

2.6. Unformatted disc support

Normally, disc storage is not explicitly dealt with by the user. Instead, the user uses the disc through the file system, which performs allocation and release of space, and lets the user work with named files rather than absolute addresses. To allow interchange of information with other systems, the Disc Executive also allows the user unformatted access to configured disc storage. For each disc unit in the system, a corresponding device file exists. For unit "x", this file will be named "DISC$x.DEV". This file simply consists of all the storage on the disc unit, treated as a single large file. This file may be opened like any other file, and the normal I/O calls used to access the storage on the device. The user must be extremely careful when using this feature, as the normal protection in the file system is bypassed, and it is easy to accidentally destroy the contents of a disc containing Disc Executive files.

3. Using the system from a program

The Disc Executive provides three basic services to programs running under its control: a set of system calls to perform services provided by the Executive, emulation of floating-point instructions, and a set of common subroutines used by most software in the system, and provided to reduce the size of the many programs that use them.

3.1. System calls

All system calls are made using the extended operation facility of the M9900 CPU. The XOP 1 instruction is reserved for system calls, and is referred to as JSYS (Jump to SYStem) throughout this manual. The Marinchip 9900 Assembler recognises the mnemonic JSYS for XOP 1. The operand of the JSYS instruction is a packet that contains the code for the request being made and storage for passing of parameters between the calling program and the Executive. The format of the packet depends upon the request being made, but the first byte is always the request index and the second byte is always a status returned by the Executive. A zero status always indicates normal completion of the request.

The following paragraphs will describe the system calls. In the paragraph heading, the mnemonic for the system call will be given, followed by the hexadecimal code for the request. Parameters passed to the system will appear as simple names. Parameters returned will be enclosed in parentheses.

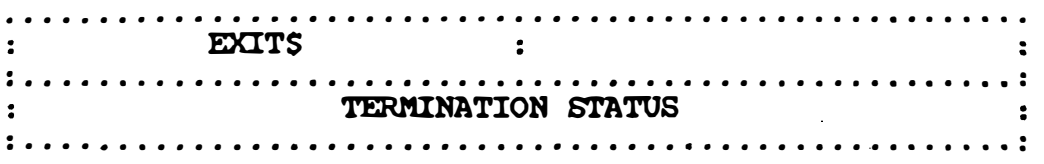
A file which defines the mnemonics for the system calls is provided by Marinchip Systems on the standard system disc. It may be included in an assembly language program by the statement:

```
COPY    "JSYS$"
```

3.1.1. Process control

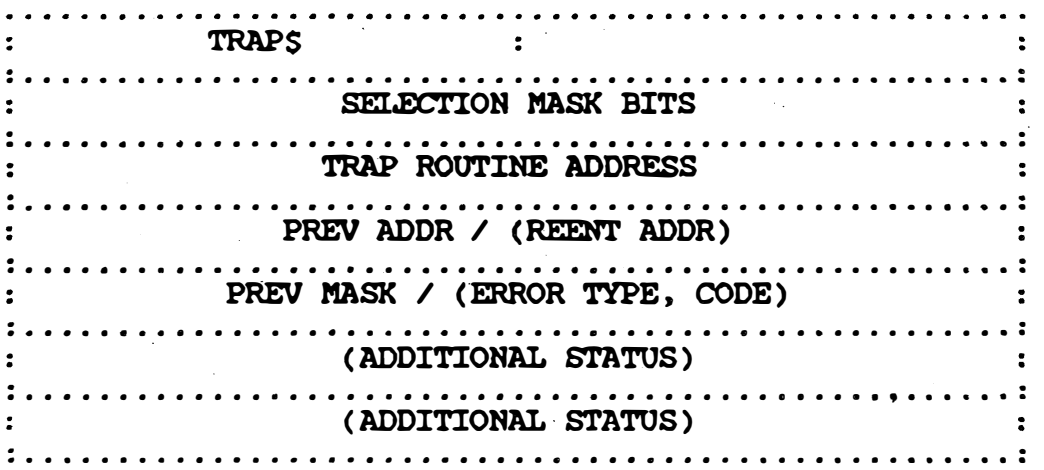
The following requests control the active program. They are a subset of the requests in the Network Operating System, which permits multiple processes in one program.

3.1.1.1. EXIT$ (02) Terminate process



The executing program is terminated and the operating system prompts the user for the next command or program. If the ⟨termination status⟩ is nonzero, a message will be printed with the termination status and the address at which the program terminated. The program's workspace registers will be dumped. Setting the ⟨termination status⟩ to a unique code and performing an EXIT$ is an easy way of indicating an error condition within a program.

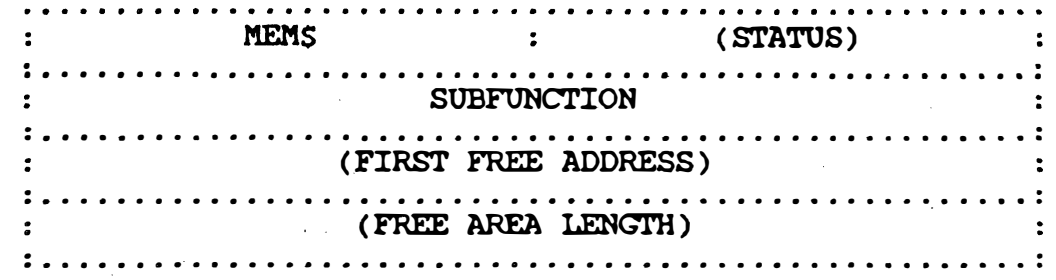
3.1.1.2. TRAP$ (0E) Reset interrupt action



The TRAP$ request allows a program to catch being interrupted by the Control C key during execution. If the TRAP$ call has not been made and Control C is pressed, the program will be terminated and its workspace registers will be dumped. If the ⟨selection mask bits⟩ are 1, the console trap will be set. Otherwise, it will be cleared. The ⟨trap routine address⟩ is where control will pass on an interrupt. When an interrupt occurs, ⟨(reent addr)⟩ will be set to the address at which the program was interrupted, and ⟨(error type, code)⟩ will be set to zero for the console trap. The first ⟨(additional status)⟩ word will be set to the address of

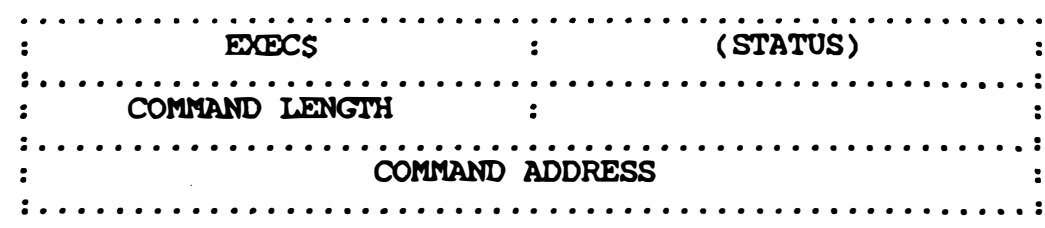
the JSYS packet in progress if the program was interrupted while a system call was in progress. The second ⟨(additional status)⟩ word is reserved for future extensions to the TRAP$ request.

3.1.1.3. MEM$ (0F) Determine memory limits



The MEM$ request may be used by a program to determine the start address and length of unallocated memory following the program itself. The ⟨subfunction⟩ must be 1 and indicates that the request is for the unallocated memory bounds. (The Network Operating System uses other subfunctions of the MEM$ request.) The first address following the program will be stored in ⟨(first free address)⟩, and the length of the free area in bytes will be stored in the ⟨(free area length)⟩ field. The ⟨(status)⟩ will be set to zero, indicating normal completion. If the ⟨subfunction⟩ is not 1, the request will be rejected and the ⟨(status)⟩ will be set to 6. This request is particularly useful in connection with the dynamic memory allocation routines described later in this manual under "System subroutines". A program can determine the size of the unused area following it in memory, establish a buffer pool in this area using the BEXP$ subroutine, and then allocate space from the pool. Such a program will automatically use all the free memory available in a system without having to be reconfigured as memory is added or removed.

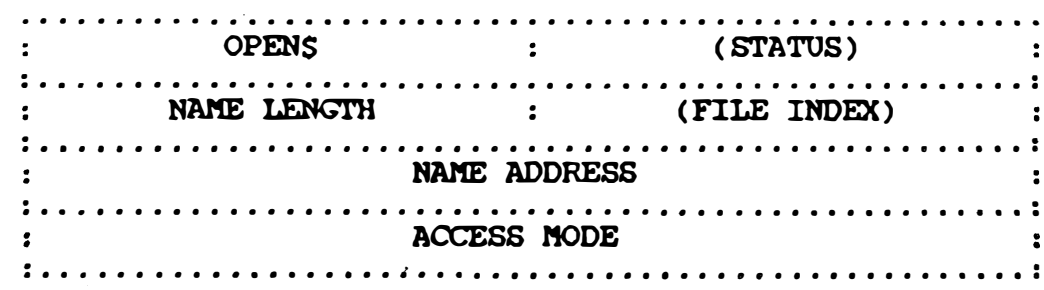
3.1.1.4. EXEC$ (11) Execute a program



The EXEC$ request allows a program to call another program. The program called overlays the calling program, so there is no return. ⟨command address⟩ is the address of an ASCII string containing the command to call the program to be executed. The format of the command is identical to what would be typed on the system console to execute the program, and may include parameters following the file name of the program to be executed. ⟨command length⟩ is the length of the command in bytes. If the EXEC$ request completes normally, the calling program will be terminated and the requested program will be loaded and executed. If an error occurs, a code indicating the nature of the error will be returned to the calling program in the ⟨(status)⟩ field, and the request will return to the calling program. A status of 3 indicates the file to be executed could not be found in the directory. A status of 5 indicates the file name specification was badly formed, and a status of 7 indicates the file named was not an executable file. Errors detected while loading the requested program will cause an error message to be printed on the system console and return to operating system command mode.

3.1.2. File control

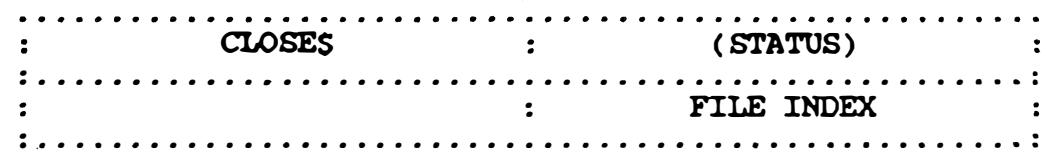
3.1.2.1. OPEN$ (05) Open a file



The address of the ASCII string containing the name of the file to

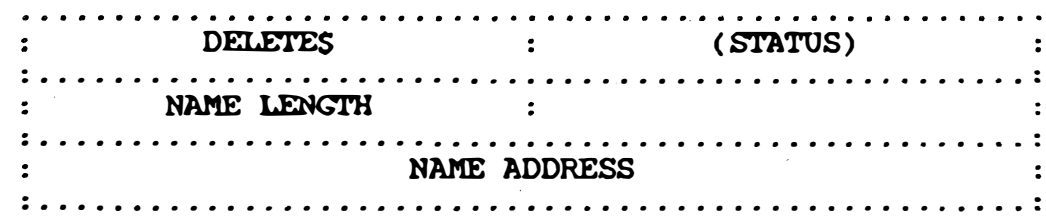
be opened is placed in ⟨name address⟩, and the length of the string is placed in ⟨name length⟩. If the file is found and opened normally, ⟨(status)⟩ will be set to zero, and ⟨(file index)⟩ will be set to the index used in all subsequent references to the file. The ⟨access mode⟩ field determines how the file may be subsequently accessed. If zero, both reading and writing will be permitted. If 1, only writing will be permitted, and if 2, only reading will be permitted. If the named file cannot be found in the directory, ⟨(status)⟩ will be set to 3. If the file name is badly formatted, ⟨(status)⟩ will be set to 5. There is a limit on the maximum number of concurrently open files. This limit is specified when the Disc Executive is generated and is normally set to 10 files. If this limit is exceeded, the OPEN$ request will be rejected with a ⟨(status)⟩ of 8. Both disc and device files may be opened by this request. All files must be opened before use. A given file may be open more than once: a separate address pointer is maintained for each open instance of a file.

3.1.2.2. CLOSE$ (06) Close a file



The file with ⟨file index⟩ (the number returned when the file was opened) will be closed. If no file with ⟨file index⟩ was open, ⟨(status)⟩ will be set to 4.

3.1.2.3. DELETE$ (09) Delete a file

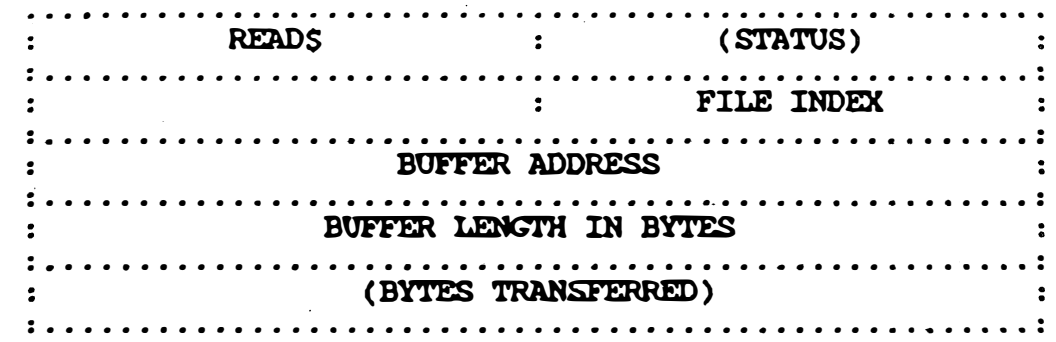


The address of the name of the file to be deleted is placed in ⟨name address⟩, and the length of the name string is placed in ⟨name length⟩. If the file was deleted properly, ⟨(status)⟩ will be set to zero. If no file of the specified name was found,

((status)) will be set to 3. If the syntax of the file name is bad, ((status)) will be set to 5. Note that a file need not be open in order to be deleted. If the file is open, it will be automatically closed.

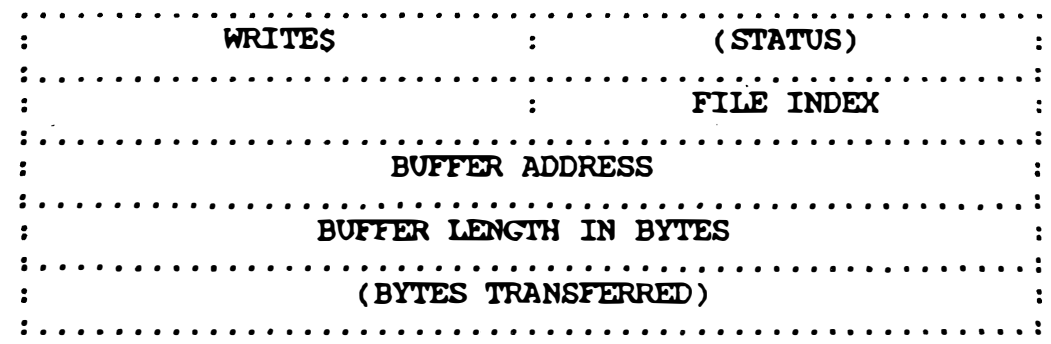
3.1.3. Input/Output

3.1.3.1. READ$ (0B) Read from a file



The next block of information from the file specified by (file index) is read into the buffer starting at (buffer address). The length of the block read is given by (buffer length in bytes). The actual length transferred is stored in ((bytes transferred)). The Disc Executive imposes the restriction that the (buffer length in bytes) must always be a multiple of 128 bytes. When reading from a device file such as the console, the actual length transferred will be the length of the logical unit of information on the device, such as a line on the console. Input from the console will end with the carriage return typed at the end of the line. The ((status)) will be zero for normal completion, 1 for end of file, and 2 if an unrecoverable I/O error occurred. If the (file index) in the packet is incorrect, the ((status)) will be set to 4. The end of file status will be given only when no information is transferred by the request. A read that starts within the file and extends past the end of the file will be truncated to the length of the information remaining in the file, and a normal status will be given. The ((bytes transferred)) field will contain the length actually delivered to the buffer.

3.1.3.2. WRITE$ (0C) Write to a file



The information starting at ‹buffer address›, with length ‹buffer length in bytes›, is written to the file specified by ‹file index›. The number of bytes actually transferred (normally the same except in the case of error) is stored in ‹(bytes transferred)›. When writing to a disc file, the Disc Executive requires that the buffer length be a multiple of 128 bytes. The values returned in the ‹(status)› field for the WRITE$ request are identical to those returned for the READ$ request (see above). Note in particular that an attempt to write with a buffer length that would extend past the end of a file will cause the length to be truncated to the length left in the file, but will still return a zero status in the packet. As a result, programs should test that the ‹(bytes transferred)› field is equal to the ‹buffer length in bytes› field after a WRITE$ request and report an error if the fields differ.

3.1.3.3. SEEK$ (0D) Set file address pointer

SEEK$	(STATUS)
SEEK BASE	FILE INDEX
OFFSET (UPPER 16 BITS)	
OFFSET (LOWER 16 BITS)	
(NEW POINTER (UPPER 16 BITS))	
(NEW POINTER (LOWER 16 BITS))	

Files are normally processed sequentially. As each READ$ or WRITE$ request is processed, the file address pointer is incremented by the number of bytes read or written to the file. To process a file randomly, the SEEK$ request may be used to set the address pointer to any desired value. The field ⟨seek base⟩ selects to what the seek is relative. If zero, the 32 bit ⟨offset⟩ field is the absolute byte number in the file. If one, the ⟨offset⟩ is added to the current position in the file to compute the new address pointer. If two, ⟨offset⟩ is relative to the end of the file. Note that ⟨offset⟩ may be positive or (two's complement) negative. At the completion of the command, the new address pointer will be stored in the ⟨(new pointer)⟩ field. The pointer may be read by doing a seek relative to the current position with an offset of zero. The Disc Executive enforces the restriction that the address pointer always be a multiple of 128 bytes.

3.1.3.4. IOCTL$ (10) Set file modes

IOCTL$	(STATUS)
STRING LENGTH	FILE INDEX
STRING ADDRESS	

The IOCTL$ request is used to set file modes. The request operates on the currently open file identified by ⟨file index⟩. The operations to be performed on the file are specified by a "function string" whose length in bytes is specified by ⟨string length⟩ and whose starting address is specified by ⟨string

address>. The function string consists of one or more type bytes, followed by data bytes in a format depending upon the type byte. The Disc Executive supports only one type byte for the IOCTL$ request, a function which sets the system console into direct (character by character) input, or returns it to normal (line buffered) mode. This operation byte has a code of 1, and is followed by one data byte. If the data byte is 1, the echoing of characters to the system console will be suppressed, and each character typed will be passed immediately to a program with a pending READ$ request from the console. If the data byte is zero, the Executive will buffer a line of input, echoing input to the terminal and providing its normal local editing facilities, then pass the entire line to the waiting program when the RETURN key is pressed. Regardless of mode, the Control C key will interrupt the executing program. The system console will be automatically reset to normal (echo) mode when a program terminates and the system command prompt appears. If the IOCTL$ operation completes normally, the <(status)> field in the packet will be set to zero. If the <file index> is for a file other than the console, the <(status)> field will be set to 4, and if the type byte is other than 1, the <(status)> will be set to 6.

3.1.4. System call error codes

When a system call (JSYS) completes normally, the <(status)> field in the request packet will be set to zero. When an error occurs, the <(status)> field is set to a numeric code indicating the error. The error numbers are common to all requests in that a given code has only one meaning regardless of which request returned it. The error codes generated by each request are discussed in the description of the request, and are summarised below.

Code	Meaning
0	Request completed normally
1	End of file on I/O request
2	Unrecoverable I/O error during request
3	File not found in directory
4	Bad file index
5	Bad file name syntax
6	Bad subfunction on request
7	File not executable
8	Too many concurrently open files

3.2. Program execution environment

When a program is given control by the Disc Executive, certain information is set up which it may retrieve by making various system calls. This section describes the execution environment of a program and how a program may determine this information at execution time.

3.2.1. Memory allocation

The standard starting address of programs run under the Disc Executive is 100 hexadecimal. Programs generated by the Linker will normally be started at this address. The area below 100 hexadecimal is reserved for the exclusive use of the Disc Executive and must not be modified by programs. The area of memory from the end of the user program to the start of the Disc Executive in high memory is available for use by the program (for example, for a buffer pool). The starting address and length of this area can be determined by use of the MEM$ system call.

3.2.2. Initial workspace

When a program is given control after being loaded by the Disc Executive, it will be given an initial set of workspace registers. This set of registers is located in an area of memory configured when the system is generated, and should be used by the user program. The user program is free to switch to other register workspaces at will with the LWPI and BLWP instructions, but use of the initial workspace allows the program to automatically adapt to the presence of a fast workspace memory area if one is available on the machine on which the program is executed.

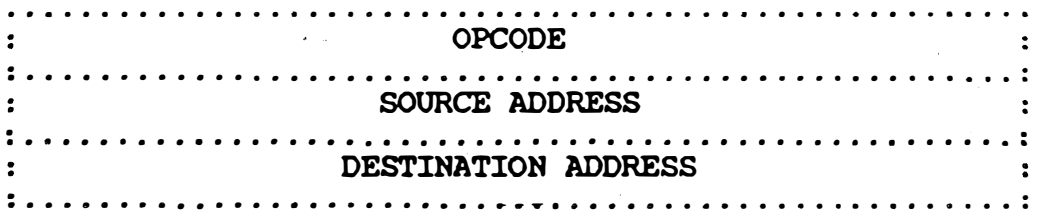
3.2.3. Program parameter string

When a program is called from the system console, parameters may follow the file name. This parameter string may be used to pass information to the program being called without having to prompt the user for the information. The system saves the parameter string (all characters following the file name) in an internal buffer, and allows the program to read it via the pseudo device file "PARAM.DEV". The file PARAM.DEV may be opened like any other file. When the READ$ request is issued on the file index returned from the OPEN$ request for PARAM.DEV, the parameter string will be

returned to the buffer address specified in the READ$ request packet. The parameter string will be terminated by a carriage return character, which will be included in the count of characters returned.

3.3. Floating point emulation

The Disc Executive provides emulation of IBM System/370 single precision floating point instructions. Emulation of instructions is requested by the XOP 2 instruction, which is defined in the Marinchip Assembler as FLOP (FLoating OPeration). The effective address of the FLOP instruction is a packet structured as follows:



The ⟨opcode⟩ field selects the function to be performed by the instruction. The action taken by different ⟨opcode⟩s is described below. The Network Operating System does not include the floating point emulation package, so programs developed for use under both the Disc Executive and the Network Operating System should use an alternate subroutine version of the floating point package, supplied by Marinchip Systems with the Network Operating System and described in its User Guide.

3.3.1.1. AE$ (01) Floating add

The floating point number at the address given by ⟨source address⟩ is added to the floating point number at ⟨destination address⟩. The result is stored at ⟨destination address⟩.

3.3.1.2. SE$ (02) Floating subtract

The floating point number at the address given by ⟨source address⟩ is subtracted from the floating point number at ⟨destination address⟩. The result is stored at ⟨destination address⟩.

3.3.1.3. ME$ (03) Floating multiply

The floating point number at ‹source address› is multiplied by the floating point number at ‹destination address›. The product is stored at ‹destination address›.

3.3.1.4. DE$ (04) Floating divide

The floating point number at ‹source address› is divided into the floating point number at ‹destination address›. The quotient is stored at ‹destination address›.

3.3.1.5. CE$ (05) Floating compare

The numbers at ‹source address› and ‹destination address› are compared, and the status bits are set depending upon their relative values. The arithmetic and logical status bits are set the same, so that either set of instructions may be used to test the result of the comparison.

3.4. System subroutines

The Disc Executive makes a set of generally useful subroutines available to programs running under its control. These subroutines are used within the Executive itself, and are provided to encourage programs to use a common set of functions for the services they provide. The subroutines are called via a system subroutine entry vector in low memory. Each location in the vector contains a jump to the actual subroutine entry point. The subroutines should always be called through the entry vector to allow them to be moved within the system from release to release.

The following table lists the entry addresses of the system subroutines. Each entry gives the entry address in hexadecimal, the mnemonic for the entry name, and a brief description of the function provided. Refer to the descriptions of the actual subroutines below for full information on how each should be called.

The mnemonics for the system subroutine entries are defined in a file provided by Marinchip Systems on the standard system disc. This file may be included in an assembly with the statement:

```
COPY      "SYSUB$"

Entry     Mnemonic  Description
-----     --------  -------------------
080       BGET      Allocate buffer
084       BGETA     Allocate buffer with error return
088       BREL      Release buffer
0E8       BEXP      Expand buffer pool

0D8       INSERT    Place buffer at end of queue
0DC       PUSH      Place buffer at head of queue
0E0       REMOVE    Remove buffer from head of queue
0E4       INITQ     Initialise queue links

08C       EDIT$     Initialise output editor
090       EDITX$    Terminate output editor
094       EDITR$    Re-enter output editor
098       ECHAR$    Edit a character
09C       ESKIP$    Skip columns
0A0       ECOL$     Tab to specific column
0A4       ECOLN$    Retrieve current column
0A8       ECOPY$    Copy text
0AC       EMSG$     Copy until stop character
0C0       EMSGR$    Continue copying after stop char
0C4       EMSG1$    Copy till stop, don't save location
0C8       EHEXF$    Edit fixed length hexadecimal
0CC       EHEXV$    Edit variable length hexadecimal
0D0       EDECF$    Edit fixed length decimal
0D4       EDECV$    Edit variable length decimal
```

The subroutines provided by the system are in three major categories as listed above: dynamic memory allocation, linked list maintenance, and output editing. Each package will be described below.

3.4.1. Calling sequence conventions

All system subroutines destroy only the registers in which results are returned, and register R11 if they are called with a BL instruction. All registers in which parameters are passed, and all registers not mentioned in the description of the subroutine may be assumed to be preserved across a call on that subroutine.

3.4.2. Output editing package

The system provides a comprehensive set of subroutines that may be used to construct messages to be read by users or placed in files. The package provides most commonly used editing functions and eliminates the duplication of effort in recoding such routines in every program written. The package is completely table-driven, and may be used to compose multiple independent messages concurrently.

3.4.2.1. Edit mode

A program wishing to use the output editing package must supply a packet containing information about the area to be edited into. The packet is 32 bytes in length. The single byte at offset 14 in the packet is the message delimiter character to be used by EMSG$, EMSGR$, and EMSG1$ (see below). The word at offset 18 in the packet is the address of the buffer where the edited output is to be placed. The length of the output buffer is placed in the word at offset 20. The rest of the packet is used by the editing routines for temporary storage, and is all the storage used by the editor: the editing package is totally reentrant. Once the packet has been defined, the program must enter edit mode.

3.4.2.1.1. EDIT$ - Enter edit mode

```
        LI      R0,<packet>
        BL      EDIT$
        <return>            R12 set to packet
```

When called, EDIT$ initialises the packet from the information supplied by the user, blank fills the output buffer, and sets the column pointer to the first character in the output buffer. The original contents of R12 is saved in the packet, and R12 is set to point to the packet. As long as the program is calling the editing routines, R12 must be left pointing to the packet.

3.4.2.1.2. EDITX$ - Terminate edit mode

```
        BL      EDITX$
        <return>            R0 = packet, R12 restored
```

The EDITX$ call terminates edit mode. Upon return, R12 will be

restored to its contents at the time EDIT$ was originally called. The address of the packet will be returned in R0. After terminating edit mode with EDITX$, the output buffer may be used in any manner desired. A subsequent call to EDIT$ will reinitialise the buffer. If desired, the user may terminate edit mode with EDITX$, do some other processing, then re-enter edit mode with EDITR$ (see below) and pick up right where he left off.

3.4.2.1.3. EDITR$ - Re-enter edit mode

```
LI      R0,<packet>
BL      EDITR$
<return>            R12 = packet
```

The EDITR$ request re-enters edit mode with a packet that has previously been left with EDITX$. The output buffer is not blanked, and the column pointer is left wherever it was at the time EDITX$ was called. Note that a packet used with EDITR$ must, at some time, have been initially set up by EDIT$: it is not possible to use EDITR$ for an initial entry to edit mode.

3.4.2.2. The column pointer

All editing done by the editing package is performed at a location defined by the "column pointer". Characters in the output buffer are numbered from zero to the number of characters in the buffer minus 1. When the package is initialised, the column pointer is set to zero, and hence points to the first character in the buffer. All of the editing subroutines store characters into the output buffer starting at the current column pointer, and advance the column pointer as they store. In addition, several routines manipulate the column pointer alone without modifying the information in the output buffer.

3.4.2.2.1. ESKIP$ - Position column pointer relative

```
LI      R0,<count>
BL      ESKIP$
<return>
```

The <count> in R0 is added to the current column position. <count> can be either positive or negative, so the pointer can be either advanced or backed up over information previously stored. Note that ESKIP$ does not blank fill the columns skipped: if

information has previously been edited into them, it will be preserved.

3.4.2.2.2. ECOL$ - Position column pointer absolute

```
        LI      R0,<column>
        BL      ECOL$
        <return>
```

The column pointer will be set so that <column> will be the next character into which information is stored. Setting <column> to zero will return to the start of the output buffer.

3.4.2.2.3. ECOLN$ - Retrieve current column number

```
        BL      ECOLN$
        <return>            R0 = column
```

Upon return from ECOLN$, user register R0 will contain the column number of the column pointer. This call is commonly used to determine the length of a line just composed with the editing routines.

3.4.2.3. Character editing

The character editing entries allow either single ASCII characters or strings of characters to be placed in the output buffer. These routines advance the column pointer as characters are stored.

3.4.2.3.1. ECHAR$ - Store single character

```
        LI      R0,<character>
        BL      ECHAR$
        <return>
```

The single ASCII character right-justified in R0 is stored in the output buffer at the current column position. The column pointer is advanced one character.

3.4.2.3.2. ECOPY$ - Copy character string

```
LI      R0,<string start>
LI      R1,<length>
BL      ECOPY$
<return>
```

The string of characters starting at the address <string start> with length <length> is copied to the output buffer. The column pointer is advanced by the number of characters stored. The <string start> address need not be aligned on a word boundary.

3.4.2.3.3. EMSG1$ - Copy string to stop character

```
LI      R0,<string start>
BL      EMSG1$
<return>
```

The string starting at <string start> is copied to the output buffer character by character until the character supplied in byte 14 of the packet passed to EDIT$ is found. This request allows a string to be specified in a manner more convenient and compact than by counting the characters in the string and using ECOPY$.

3.4.2.4. Message editing

Most messages generated by programs consist of fixed information with variable information inserted by the program. The message editing entries allow easy composition of such messages.

3.4.2.4.1. EMSG$ - Start message editing

```
LI      R0,<message address>
BL      EMSG$
<return>
```

The message starting at <message address> will be copied into the output buffer character by character until a stop character equal to the character in byte 14 of the packet passed to EDIT$ is found. The address of the character following the stop character will be saved in the packet. The column pointer is advanced once for each character stored in the buffer.

3.4.2.4.2. EMSGR$ - Continue message editing

```
        BL      EMSGR$
        (return)
```

EMSGR$ works exactly like EMSG$, except the image copied starts at the address saved by the last EMSG$ call. EMSGR$ copies to the next stop character, then saves the address of the character following the stop character. EMSG$ and EMSGR$ allow portions of a message to be copied, pausing periodically to insert information in the message using the other editing routines.

3.4.2.5. Numeric editing

The editing package includes entries to edit 16 bit numbers to either hexadecimal or decimal. Both variable length and fixed length editing is provided.

3.4.2.5.1. EDECV$ - Variable length decimal edit

```
        LI      R0,(value)
        BL      EDECV$
        (return)
```

The value in R0 will be edited as a decimal integer. If the sign bit is set, a minus sign will be edited before the number. EDECV$ edits only the number of characters required to hold the number edited to decimal: for example, the number 1 would occupy one character, 234 would require three, and -16255 would require six. The column pointer will be left set after the last digit edited.

3.4.2.5.2. EDECF$ - Fixed length decimal edit

```
        LI      R0,(value)
        LI      R1,(length)
        BL      EDECF$
        (return)
```

The value in R0 is edited right-justified in a field whose width is specified by R1. The column pointer is left after the last digit edited. If the number supplied in R0 requires more characters to edit than the field size contains, it will overflow the field to the right. Characters in the field into which digits

are not edited will be unchanged: hence it is possible to edit with leading zeroes or check protection by pre-editing the desired fill into the field, backing up with ESKIP$ or ECOL$, then overlaying the number in the field with EDECF$.

3.4.2.5.3. EHEXV$ - Variable length hexadecimal edit

```
        LI      R0,<value>
        BL      EHEXV$
        <return>
```

The value passed in R0 is edited to hexadecimal as an unsigned 16 bit integer. If the value in R0 is larger than 9, a leading zero will be edited, following the system convention that a leading zero signifies hexadecimal. The column pointer will be left after the last digit edited.

3.4.2.5.4. EHEXF$ - Fixed length hexadecimal edit

```
        LI      R0,<value>
        LI      R1,<length>
        BL      EHEXF$
        <return>
```

The value passed in R0 is edited right-justified in a field with length passed in R1. All characters in the field before the first nonzero digit of the edited number will be filled by zeroes. The column pointer will be left immediately following the last digit edited. If the value is too large to fit in the field size supplied, the high-order digits will be truncated. This means, for example, that the low byte of R0 may be edited simply by supplying a count of 2 in R1.

3.4.2.6. Sample use of the editing package

The following program fragment uses the editing routines to build an error message as might be generated by a compiler. Note how the various routines are used to insert specific information into the "canned" message text.

```
        LI      R0,EPKT         Load editor packet address
        BL      EDIT$           Start up the editor
        LI      R0,ERRMSG       Load error message address
        BL      EMSG$           Copy message
```

```
        MOV     LINENO,R0       Load line number of error
        BL      EDECV$          Edit it to decimal
        BL      EMSGR$          Copy to value
        MOV     BADVAL,R0       Load the bad value
        LI      R1,4            Load length to edit
        BL      EHEXF$          Edit value to hexadecimal
        BL      EMSGR$          Copy rest of message
        BL      ECOLN$          Get number stored
        MOV     R0,OUTLEN       Save output message length
        BL      EDITX$          Terminate the editor
        .       .
        .       .
        .       .
EPKT    BSS     14              Editor packet
        BYTE    '&',0           Stop character and fill
        BSS     2
        DATA    OUTBUF,80       Output buffer and length
        BSS     10
.
OUTBUF  BSS     80              Output buffer
.
ERRMSG  TEXT    'Error on line &.  Bad value &.&'
```

3.5. Storage and linked list subroutines

The dynamic memory allocation and linked list subroutines share a common workspace area and calling sequence conventions. As a result, they will be discussed together here. In order to use these routines, the user must provide a workspace area and buffer pool control storage. This area is formatted as follows in an assembly program:

```
BHEAD   DATA    BHEAD,-1,BHEAD,BHEAD Buffer pool head
.
PWS     EQU     $-16            Primitive work space tag
        BSS     4               Space for R8, R9
        DATA    BHEAD           Storage head pointer
        BSS     10              Space for R11 - R15
```

The various routines are entered via the BLWP instruction through a set of context switch vectors supplied by the user. These vectors reference the workspace defined above, and the entry point to the proper subroutine name. The entry vectors are commonly given the same name as the subroutine name, but followed by a dollar sign. A definition for an entry vector for all the buffer allocation and linked list routines is as follows:

```
INSERT$ DATA    PWS,INSERT
```

```
PUSH$     DATA      PWS,PUSH
REMOVE$   DATA      PWS,REMOVE
INITQ$    DATA      PWS,INITQ
BGET$     DATA      PWS,BGET
BGETA$    DATA      PWS,BGETA
BREL$     DATA      PWS,BREL
BEXP$     DATA      PWS,BEXP
```

A workspace area and entry vector, formatted as given above, is supplied by Marinchip Systems in the file "PRIMW$" on the standard system disc, and may be included in an assembly with the statement:

```
          COPY      "PRIMW$"
```

3.5.1. Dynamic memory allocation routines

The dynamic memory allocation routines maintain a pool of free space, allocating buffers from it, releasing them back to it, and allowing space to be added to the pool at any time. The allocator uses a free list chain technique which allows buffers to be allocated with the size the user requested, and does not limit the user to a potentially wasteful power of two size as do many "buddy system" schemes. The overhead storage used to control the buffers allocated amounts to only eight bytes per buffer. When space is released and the adjacent space is an available buffer, it is combined into one large area, so that fragmentation problems are minimised.

3.5.1.1. BEXP$ - Add space to buffer pool

```
          LI        R0,<length of area to add>
          LI        R1,<address of area to add>
          BLWP      BEXP$
          <return>
```

The buffer pool defined in the initial workspace for the allocation routines is void: no free space is provided. Before allocation may begin, the user must supply the raw pool of storage from which buffers are to be allocated. This is done with the BEXP$ call. The area passed is typically the area from the end of the code portion of the program to the end of system memory, hence all free memory is automatically available for buffers. R0 should contain the length of the area in bytes, and R1 should point to the first byte in the area to be added to the buffer pool: neither need be even. BEXP$ can be called at any time to add

additional storage to the buffer pool. For example, some programs initially define their buffer pool with BEXP$, then after all their initialisation is complete, release the area occupied by the initialisation code itself into the buffer pool.

3.5.1.2. BGET$ - Allocate a buffer: error if none

```
        LI      R1,<size in bytes>
        BLWP    BGET$
        <return>                        R1 = buffer allocated
```

The BGET$ entry will allocate a buffer of the requested size and return its address in R1. If there is insufficient space to allocate a buffer of the requested size, the program will be terminated with an error code of 010. Programs which wish to handle the out of buffers situation themselves should use the BGETA$ request, described below. Note that buffers allocated by BGET$ will always start on a word boundary.

3.5.1.3. BGETA$ - Allocate a buffer: return if none

```
        LI      R1,<size in bytes>
        BLWP    BGETA$
        DATA    <insufficient space>
        <return>                        R1 = buffer allocated
```

A buffer will be allocated with the size requested in R1 and its address will be returned in R1. If insufficient storage remains to allocate a buffer of the requested size, the routine will return at the address specified for <insufficient space>. Buffers allocated by BGETA$ will always start on a word boundary.

3.5.1.4. BREL$ - Release buffer

```
        LI      R1,<buffer address>
        BLWP    BREL$
        <return>
```

The BREL$ entry returns a buffer allocated by BGET$ or BGETA$ to the available space pool. The address passed in R1 on the call to BREL$ must be an address previously returned by BGET$ or BGETA$. To add storage outside the buffer pool to it, use the BEXP$ request, documented above.

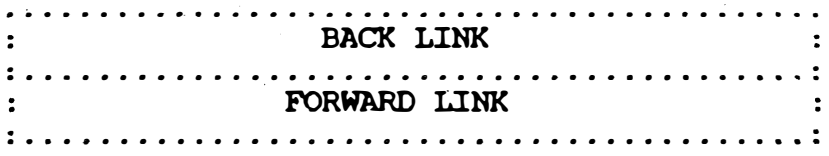
3.5.1.5. Buffer allocation errors

The buffer allocation routines will terminate the requesting program if certain errors are detected. The error code used to terminate the program indicates which error was detected. The following are the error codes generated by the buffer allocation routines:

010	No space for buffer on BGET$. This error causes an abnormal return to the program if BGETA$ is used instead of BGET$.
011	Attempt to release unallocated buffer via BREL$. Check address passed to BREL$.
012	Backpointer in next buffer was bad. This will result if the program using the buffer stored off the end of the buffer, and may also result if a bad address if passed to BREL$.

3.5.2. Linked list routines

The following subroutines manipulate doubly linked lists of buffers. Each list is defined by its "list head", which is a two word (four byte) block of storage arranged as follows:



The back link points to the last buffer on the queue, and the forward link points to the first buffer on the queue. If there is only one buffer on the queue, the forward and back links will both point to that buffer. If the queue is empty, both links will point to the address of the queue head itself. Buffers to be placed on the queue must have a two word area at the start reserved for queue links. The link area at the start of the buffer will be used for back and forward links exactly like those in the queue head. Storage after the link area may contain anything the user desires, and is in no way examined or manipulated by the queue routines.

3.5.2.1. INITQ$ - Initialise queue links

```
        LI      R9,<queue>
        BLWP    INITQ$
        <return>
```

The links in the two word area whose address is passed in R9 will both be set to point to the address in R9. This initialises an area of storage as an empty queue. This can also be easily done by user code, and is provided only as a convenience and to encourage dynamic creation of queue heads.

3.5.2.2. INSERT$ - Insert buffer at queue end

```
        LI      R8,<buffer>
        LI      R9,<queue>
        BLWP    INSERT$
        <return>
```

The buffer whose address is passed in R8 is chained at the end of the queue whose head address is passed in R9. Only the links in the first two words of the buffer pointed to by R8 will be changed.

3.5.2.3. PUSH$ - Insert buffer at queue start

```
        LI      R8,<buffer>
        LI      R9,<queue>
        BLWP    PUSH$
        <return>
```

This entry is identical to the INSERT$ entry described above, but the buffer is placed at the start of the queue instead of the end. A buffer placed on a queue with PUSH$ will always be the first to be removed by a subsequent call on REMOVE$.

3.5.2.4. REMOVE$ - Remove next buffer from queue

```
        LI      R9,<queue>
        BLWP    REMOVE$
        <return>                        R8 = buffer
```

The first buffer on the queue will be removed from the queue and

its address will be returned to the user in R8. The address returned will be the address of the first link word in the buffer, which is the same address passed to INSERT$ or PUSH$ when the buffer was placed on the queue. If the queue was empty, R8 will contain the address of the queue head itself upon return. This allows the empty condition to be tested simply by comparing the address returned in R8 with the queue address still in R9. Hence, a remove with empty test would be coded as follows:

```
LI      R9,MYQUEUE      Load queue address
BLWP    REMOVE$         Remove next buffer
C       R8,R9           Was queue empty ?
JEQ     EMPTY           Yes.  Don't do anything
.       .
.       .
.       .
```

4. System utility programs

The Disc Executive supports a wide variety of software packages, including compilers, assemblers, debug packages, and utilities. This section of the manual will describe all of the standard programs which are called by commands from the system console. For many commands, this documentation is complete. For complex software packages such as the assembler or Pascal compiler, a brief command description is included and the user is referred to the appropriate manual for further information.

Several of the utility programs described in the following sections are actually different names for a common program called, for historical reasons, "the shell". This program, which is stored in the file SHELL$.OBJ, is automatically called by the Disc Executive when any of the commands it implements are entered. The fact that all of the vital file-oriented commands are performed by a single program makes the task of setting up new system discs much more simple, as only that program need be copied onto a new disc. The commands currently implemented in SHELL$.OBJ are BCOPY, CREATE, DELETE, DIRECT, PREP, and RENAME.

4.1. ASM - Assembler

The Marinchip Assembler is an expression-oriented relocatable assembler for the Marinchip 9900 computer. It accepts a source syntax largely compatible with the Texas Instruments 9900 assembler, and produces relocatable code completely compatible with that used by Texas Instruments.

4.1.1. Calling the assembler

The assembler is called with a command of the form:

ASM ‹reloc›=‹source›[,‹listing›]

where ‹source› is the name of the file containing the source program to be assembled, ‹reloc› is the name of the file in which the relocatable output of the assembler is to be stored, and ‹listing› is the optional file where the assembly listing is to be written. If no ‹listing› file is specified, no listing will be generated, but lines with assembly errors will still be listed on the system console. If the assembly listing is sent to a disc or device file other than the console, lines with errors will still be logged to the console.

4.1.2. For more information

Refer to the manual "Marinchip 9900 Assembler User Guide" for complete information on writing assembly language programs and using the assembler.

4.2. BASIC - BASIC interpreter

Marinchip BASIC is a comprehensive implementation of the BASIC language, with extensions for string processing, file access, and interface to hardware devices. BASIC precompiles the program to speed execution speed, and automatically operates in integer or floating point mode as required by the program. Marinchip BASIC provides immediate execution of statements, a symbolic statement trace, and the ability to pause execution, modify a program, and resume it. These features greatly ease the debugging of complex programs.

The BASIC present in a given system may be either the standard Marinchip BASIC, or Extended Commercial BASIC, an optional software package which also provides 16 digit decimal accuracy for numbers, random access files, CHAIN between programs with common variables, and the ability to save precompiled code files and execute the under a special runtime system which occupies less memory than the complete interpretive BASIC. Consult the person responsible for software maintenance at your installation to determine which BASIC is available on the machine you use.

4.2.1. Calling BASIC

BASIC is called by simply typing its name:

```
BASIC
```

When loaded, it will issue a command prompt ")", and await a command. The user can either enter a program, load a previously written program, or use BASIC as a desk calculator by entering BASIC statements without line numbers.

4.2.2. For more information

Refer to the manual "Marinchip 9900 BASIC User Guide" for complete documentation of the BASIC language and the Marinchip implementation.

4.3. BCOPY - Binary file copy

BCOPY performs a binary (transparent) copy between two files. It permits data to be transferred regardless of its content, and thus allows creation of an exact copy of the input. BCOPY is invoked by a command of the form:

BCOPY ⟨output⟩=⟨input⟩,⟨number⟩

⟨output⟩ is the name of the output file and ⟨input⟩ is the name of the input file. BCOPY will copy the data from the input file to the output file. If the optional ⟨number⟩ specification is supplied, the copy will be terminated after ⟨number⟩ blocks of 128 bytes have been copied. If ⟨number⟩ (and the preceding comma) are omitted, the copy will continue until either the end of the input file, the end of the output file, or an I/O error occurs. BCOPY will always print the number of blocks copied, and will indicate the reason for termination of the copy, unless the reason was the satisfaction of a ⟨number⟩ specification.

4.3.1. Examples of use

To copy a file PROG1 into a file called BKPG1, BCOPY would be used as follows:

BCOPY BKPG1=PROG1

To copy the first 10 blocks of the file SAVFIL into the file MYPROG, one would use:

BCOPY MYPROG=SAVFIL,10

4.3.2. Restrictions - device files and BCOPY

BCOPY may be used without restriction on disc files. When it is used with device files, the user must be aware of the fact that input from many device files (for example, the system console) is rarely the standard block length, 128. Since the Disc Executive requires blocks written to a disc file to be multiples of 128 bytes, BCOPY cannot be used to transcribe from a device file to a disc file. Transfers in the other direction (disc file to device file) will cause no problems. TCOPY, the text copy utility, will serve for most of the applications where transcription from a device file to a disc file is required. TCOPY is also more efficient for such applications since it buffers the input into

128 byte blocks. See the section later in this manual describing TCOPY for more information.

4.3.3. Messages

Error: Specify ⟨ofile⟩=⟨ifile⟩,⟨number⟩
This message is given when the parameters to BCOPY are bad or omitted.

⟨number⟩ blocks copied.
This message will appear at the end of the copy to indicate the number of (128 byte) blocks copied.

Copy terminated by end of input file.
This message is issued when the end of the input file is reached. Note that this message will appear when the input and output files are the same size.

Copy terminated by error reading input file.
The operating system has returned an error status on a read of the input file.

Copy terminated by end of output file.
This message is issued when the end of the output file is reached and information remains to be copied from the input file. The user should be careful that no valid information was lost in the truncation.

Copy terminated by error writing output file.
The operating system has returned a nonrecoverable error writing a block to the output file.

File ⟨filename⟩ does not exist.
The named file (input or output) could not be found in the file directory.

4.4. BRAINS - BRAINSTORM diagnostic package

BRAINSTORM is a comprehensive processor and memory diagnostic developed by Marinchip Systems for the Marinchip 9900 computer. BRAINSTORM includes both confidence tests, which test the computer under a simulated worst-case program situation, and diagnostic tests, which aid in the isolation of specific problems and their correction.

4.4.1. Running BRAINSTORM

BRAINSTORM runs under any Marinchip operating system, and uses the operating system for all of its I/O. As a result, the diagnostic need not be reconfigured when system peripherals change. The package itself occupies the memory between 100 and 2000 hex, so any area after 2000 is available for memory testing. BRAINSTORM is invoked from operating system command mode simply by typing the file name containing the program. In standard released systems, this file is named "BRAINS". Following the file name is a parameter that specifies the test to be run. The format of the parameter is a single character test identifier, an equal sign, and a list of parameters specific to the test selected. The available tests are as follows:

M	Memory diagnostic
P	Processor (CPU) diagnostic

4.4.1.1. Memory diagnostic

The memory diagnostic is invoked by the parameter string:

M=⟨start addr⟩,⟨bytes to test⟩,⟨passes⟩

where ⟨start addr⟩ is the first address to test, ⟨bytes to test⟩ is the length of the area to be tested in bytes, and ⟨passes⟩ is the number of times the test is to be run before automatically terminating. The ⟨start addr⟩ and ⟨bytes to test⟩ specification will be rounded down to even word addresses if odd addresses are specified. For example, assuming BRAINSTORM is in the standard file name "BRAINS", and you wished to test 1000 hex bytes (4096 decimal) starting at address 6000 hex, and you wished the test to iterate 50 times, the command typed in to the operating system would be:

```
BRAINS M=6000,1000,50
```

Note that the first two parameters are automatically scanned as hexadecimal, and the third is automatically scanned as decimal.

Before starting the test, the parameters will be confirmed by the message:

Brainstorm now testing 6000 through 6FFF, 50 times.

If an error is detected, a two line error message will appear of the form:

Error in memory test ‹test description›
Address ‹fail addr›: Expected ‹good›, received ‹bad›.

The ‹test description› is the number and name of the specific subtest that failed (see below). The ‹fail addr› is the address which failed. ‹bad› is what was read from the address, and ‹good› is what was expected by the test.

At the end of each pass through the test, the message:

End pass ‹pass›.

will appear. If any errors occurred on this pass of the test, the message:

‹count› errors.

will be appended to the "End pass" message. If any errors have occurred earlier in this execution of BRAINSTORM, whether on the most recent pass or not, the message:

Total errors ‹count›.

will appear at the end of the "End pass" message.

4.4.1.1.1. Memory subtests

The following paragraphs describe the subtests performed by BRAINSTORM. One pass through the memory test consists of running each subtest once, in the order listed below.

4.4.1.1.1.1. 1A: Clear to zero

Each word in the test area is cleared to all zero bits, then immediately read back and tested against zero. Failure to clear

is failure of this test.

4.4.1.1.1.2. 1B: Set to all ones

Each word in the test area is set to all one bits, then immediately read back and tested against all ones. Failure of all bits to set is considered a failure.

4.4.1.1.1.3. 2A: Sliding one bit

A pattern of a single one bit with all other bits zero is written through each word in the test area. Each word is read back after being written and tested against the pattern written. Failure to compare is a failure of the test. After all words in the test area have been completed, the pattern is shifted one bit right, and the test is performed again. The test starts with the pattern 8000 hex and completes with 0001 hex.

4.4.1.1.1.4. 2B: Sliding zero bit

A pattern of a single zero bit with all other bits one is written through each word in the test area. As each word is written, it is immediately read back and tested against the pattern stored. After all words have been tested, the pattern is shifted circularly one bit right, and the test continued until all 16 possible patterns have been tested. The test starts with the pattern 7FFF hex and ends with the pattern FFFE.

4.4.1.1.1.5. 3: Address interference test

This test is intended to detect shorted address lines and failing address decode hardware in memories. Each word in the test area is tested. To test a specific word, it is set to the hex pattern 1234. Then each address bit in the address of the word under test is inverted. If the address generated by inverting the bit is still within the test area, the pattern DEAD is stored in that address. After all possible addresses within the test area generated by inverting bits of the original address have been set to the pattern DEAD, the original word is read back and tested. If it has been changed from the original value of 1234, the address interference test has failed. The test is repeated until all words in the test area have been tested. This test is most

effective if run over the entire addressing range of a memory component, as excluding even a small region will eliminate some possibly defective address bits from the scrutiny of this subtest. If this test fails, the problem is almost certainly a shorted address lead or other decoding error that is mapping two different addresses into the same memory cell. Careful examination of the error messages generated by this test should lead to the specific failing component. (The output from the next subtest, Addressing Validation, may also be useful).

4.4.1.1.1.6. 4: Addressing validation

The addressing validation test simply writes the address of each location in the test area in the cell at that address. After all locations have been so set, they are read back and tested to contain their own address. This subtest detects addressing failures more subtle than those detected by the Address Interference test above.

4.4.1.1.1.7. 5: Byte addressing

This subtest writes an ascending value, modulo 256, into all bytes in the test area. When all bytes have been set, the area is read back byte by byte and tested against the expected value. Since byte addressing is performed in the M9900 processor itself by masking the 16 bit data, this is more of a processor test than a memory test. It is included since it may detect particular memory timing problems that only appear in the case of byte addressing.

4.4.1.2. Processor diagnostic

The processor diagnostic is invoked by the parameter string:

P=⟨passes⟩

where ⟨passes⟩ is the number of times the test is to be repeated before terminating. The diagnostic will test various internal operations of the processor in each pass of the test, and type an "End pass." message at the end, exactly like the Memory diagnostic (see above). If a failure is detected, a message of the form:

Error in CPU test: ⟨type⟩ instruction failure.

will be typed, and that pass of the test will be immediately

terminated. ⟨type⟩ describes the subtest that failed. The possible values of ⟨type⟩ are:

Basic shift/AND/OR
BLWP/RTWP/status register
ABS
Add
Add bytes
INC/DEC
SWPB
Multiply
Divide
Jump odd parity
SZC/SZCB
SOCB

These refer to the instruction whose failure most likely led to the failure of the subtest. Since the entire arithmetic and logical processor is integrated onto a single IC, a failure of the CPU test generally indicates that the CPU chip must be replaced. Bad memory, however, may often cause the CPU test to fail, so CPU chip failure (a VERY rare occurrence) is indicated only when the memory diagnostic runs without error and the CPU test fails.

4.5. CREATE - Create a file

CREATE ⟨file⟩,⟨size⟩
or CRE ⟨file⟩,⟨size⟩

The CREATE command creates a file on a disc. If the ⟨file⟩ named includes a ⟨unit⟩ specification, the file will be placed on that unit. Otherwise, the file will be allocated on the system disc. The ⟨size⟩ is expressed in terms of 128 byte sectors. CREATE will scan the directory of the unit on which the file is being placed, locate the "best fit" for the size requested, and enter the file in the directory of that unit. If a file by the same name already exists, it will be deleted automatically. If there is no free block on the unit large enough to hold the requested file size, the message:

Insufficient contiguous space for file.

will be issued and the CREATE command will be ignored. If a DIRECT reveals that there is enough total space for the file but no single block large enough, the PACK command may be used to recover the fragmented space, then the CREATE will succeed.

4.6. DELETE - Delete file or group of files

The DELETE utility may be used to delete a single file, or may be used to delete a group of files based on selection criteria supplied by the user. DELETE is invoked by a command of the form:

```
   DELETE ⟨file⟩,...
or DELETE ⟨selection⟩,...
or DEL ...
```

If called with a conventional file name, DELETE will delete that single file. If the specification contains the characters "?" or "*", it is taken as a selection specification designating a group of files. If a character of the ⟨selection⟩ specification is "?", files with any character in that position will be processed. A specification of the form:

```
   NAME.*
```

will choose all files with NAME before a period in a file name and any text after the period, while specifiying:

```
   *.TYP
```

chooses all files with any name and the text TYP after the period in the name. To choose all files on a volume, any of the following specifications may be used:

```
   ⟨unit⟩/
   ⟨unit⟩/???????????
or ⟨unit⟩/*.*
```

When a ⟨selection⟩ specification is used, DELETE will prompt the user with each file name about to be deleted. If the user answers the prompt with "Y", the file will be deleted. If the prompt is answered with "N", the file will not be deleted.

4.7. DIRECT - List file directory

The DIRECT utility lists file directories. DIRECT may be used to list the directory entry for a single file, all files on a disc, or groups of files chosen by masking their names. The directory listing may be either typed on the system console, or sent to a printer. DIRECT may be invoked with any of the following specifications:

```
   DIRECT ‹unit›/,...
   DIRECT ‹file›,...
or DIRECT ‹selection›,...
or DIR ...
```

The first form of the command lists all files on the specified disc unit. Three files will be listed on each line of output from DIRECT, allowing more files to be seen before scrolling off the top of a display.

The second form of the command, naming a file, will list the directory item for the named file.

The third form of the command specifies a ‹selection› which names a group of files for which the directory items are to be listed. If a character of the ‹selection› specification is "?", files with any character in that position will be processed. A specification of the form:

```
   NAME.*
```

will choose all files with NAME before a period in a file name and any text after the period, while specifiying:

```
   *.TYP
```

chooses all files with any name and the text TYP after the period in the name.

Regardless of the form of call used, the item for each file will be printed in the format:

```
   FILENAME  size  start
```

where "FILENAME" is the file name, "size" is the file size in sectors, and "start" is the starting block number on the disc. When listing all files on a unit, a summary line will be printed giving the total free space and the largest contiguous free block available on the unit.

Multiple specifications may be given to DIRECT, separated by commas. The action of DIRECT is the same as if each specification were given on a separate DIRECT command. If any specification is preceded by a plus sign, "+", the listing generated by that specification will be sent to the print device, PRINT.DEV. This permits printed directory listings to be made for later reference. For example, to print the directories of both unit 1 and unit 2, one might use:

```
DIRECT +1/,+2/
```

4.8. DU - Disc utility

The Marinchip Disc Utility provides the functions necessary to test, format, dump, patch, and prepare floppy discs for use with Marinchip operating systems. Two versions of the Disc Utility are available. The standard version, supplied with the Disc Executive release disc, uses the disc handler within the Disc Executive. This allows the Disc Utility to be smaller, and usable on any system without special configuration, but restricts its ability to perform I/O functions not available through the system (such as formatting discs). Special versions of the Disc Utility containing handlers for specific disc devices can be ordered from Marinchip Systems. Contact your dealer or Marinchip Systems for price and ordering information.

4.8.1. Using the disc utility

The Disc Utility is loaded and executed simply by typing its name, DU, to the operating system. The Disc Utility will be loaded and will prompt the user for a command with an asterisk (*). At this time, any disc utility command may be typed. At the completion of each command, the prompt will reappear. When you have finished with the disc utility, enter the command "END". It will exit to the operating system.

4.8.1.1. Disc utility commands

All disc utility commands are one or two characters in length. Any number of spaces may precede the command name, and at least one space must follow the command name if any parameters follow. In the following command descriptions, the parameters expected will be enclosed in corner brackets. The format of the parameters is as follows:

⟨disc⟩ This parameter is the disc number. Discs in the system are numbered starting from one through the highest numbered disc in the system.

⟨track⟩ This parameter is the track number on the selected disc. Tracks are numbered from 0 to 76, for a total of 77 tracks on each disc.

⟨sector⟩ This parameter is the sector number within the selected disc and track. Sectors, for some strange reason, are numbered from 1 to 26. One of the most common parameter

errors is trying to reference sector zero. There is no sector zero!

Wherever a specification of the form:

⟨disc⟩,⟨track⟩,⟨sector⟩

is used to identify a specific sector, the alternate construction:

⟨disc⟩.⟨block⟩

may be used. The ⟨block⟩ refers to the absolute sector number on the disc, with the first sector considered as zero. This alternate specification form can be useful when using the Disc Utility on Disc Executive formatted discs, as the file directory addresses sectors by block number, rather than track and sector numbers.

4.8.1.1.1. A - Dump in ASCII

A ⟨start byte⟩,⟨word count⟩

The ASCII dump command dumps the contents of the sector buffer in ASCII. If all parameters are omitted, the entire buffer will be dumped. If a start byte is specified, the word containing that byte will be dumped. If both a start byte and a length are specified, the number of words requested will be dumped starting with the selected byte. The sector buffer is read by the "R" command and written by the "W" command, both described below.

4.8.1.1.2. CD - Copy disc

CD ⟨disc⟩ ⟨disc⟩

This command copies the entire contents of the first disc to the second disc. It is a fast and effective way to back up the contents of one disc on another. Any data previously stored on the second disc will be destroyed. This command requires the user to confirm that it is OK to wipe out the data on the second disc. If the command:

```
CD 1 3
```

is typed, the question:

```
Really want to destroy data on disc 3?
```

will appear. This must be answered "yes" before the operation will begin. Any other answer will cause the command to be ignored.

4.8.1.1.3. CT - Copy track

CT ‹disc›,‹track› ‹disc›,‹track›

This command copies an entire track from the first ‹disc›,‹track› to the second. Note that the source and destination tracks may be different.

4.8.1.1.4. D - Dump in hexadecimal

D ‹start byte›,‹word count›

The Dump command dumps the contents of the sector buffer in hexadecimal. If all parameters are omitted, the entire buffer will be dumped. If a start byte is specified, the word containing that byte will be dumped. If both a start byte and a length are specified, the number of words requested will be dumped starting with the selected byte. The sector buffer is read by the "R" command and written by the "W" command, both described below.

4.8.1.1.5. END - End disc utility

END

The End command causes the Disc Utility to terminate. Control will return to the operating system.

4.8.1.1.6. N - Read and dump next sector

N

The sector following the last sector read by an R, RA, or RD command is read and dumped. The sector is dumped in the format last used to dump a sector. The N command is primarily used when reading through a disc looking for some particular data. The address of the sector being read will be printed before the sector is dumped.

4.8.1.1.7. PA - Patch buffer

PA ‹start byte›

This command allows the contents of the sector buffer to be patched. If ‹start byte› is omitted, zero is assumed. The command will display the current offset and the contents of the word at that offset. If a carriage return is typed, the next word will be displayed. If a number is entered, it will replace the word at the current location. An up-arrow (^) will cause the previous word to be displayed, and a right corner bracket (›) followed by a number will set the offset to that byte address. Numbers entered for this command will be assumed decimal unless a leading zero appears before the number, in which case hexadecimal will be assumed. Entering an at sign (@) will stop the Patch command and return the user to normal command level.

4.8.1.1.8. R - Read into buffer

R ‹disc›,‹track›,‹sector›

This command reads the selected sector into the sector buffer. Once read in, the data may be dumped by the "D" command, patched by the "PA" command, and written back out by the "W" command.

4.8.1.1.9. RA - Read and dump in ASCII

RA ‹disc›,‹track›,‹sector›

This command reads the selected sector into the sector buffer and then dumps it in ASCII. The action of this command is identical to an "R" command followed by a "A" command.

4.8.1.1.10. RD - Read and dump in hexadecimal

RD ‹disc›,‹track›,‹sector›

This command reads the selected sector into the sector buffer and then dumps it in hexadecimal. The action of this command is identical to an "R" command followed by a "D" command.

4.8.1.1.11. VD - Validate disc

VD ‹disc›

This command reads every sector on the selected disc. If any errors occur, they will be logged and the command will continue. This command is intended for incoming inspection of new discs and periodic checking to make sure that no bad sectors are lurking on a disc.

4.8.1.1.12. VT - Validate track

VT ‹disc›,‹track›

This command is identical to the Validate Disc command described above, but only one selected track is validated.

4.8.1.1.13. W - Write

W ‹disc›,‹track›,‹sector›

The data in the sector buffer are written out to the selected sector.

4.8.1.1.14. WB - Write back

WB

The data in the sector buffer are written back to the sector from which they were originally read with the R, RA, RD, or N command. The WB command may be used only if no intervening command which reads into the sector buffer has been used since the sector was originally read. The WB command is normally used to write back data read in with the R command, then patched via the PA command.

4.9. EDIT - Text editor

The Marinchip Text Editor (EDIT) is a line-oriented context editor based on the Project MAC editor originally developed at MIT. The editor offers powerful interactive editing taking full advantage of the full-duplex terminal support and instantaneous response offered by the Disc Executive. The editor uses the file system to automatically page files larger than memory to disc to allow files much larger than system memory to be edited without explicit user effort.

4.9.1. Calling the editor

The most general form of call on the editor is:

EDIT ⟨output file⟩=⟨input file⟩

Either or both of these file names may be omitted, with results illustrated by the examples given below.

EDIT MYFILE	Reads in MYFILE, and stores output back in MYFILE.
EDIT NEW=	Creates file NEW from text entered from the console.
EDIT =LISTNG	Reads in file LISTNG to be examined, but not updated.
EDIT NEW=OLD	Reads in file OLD, stores updated output in file NEW.
EDIT	Gives user complete control over input and output handling via editor commands.

4.9.2. Using the editor

A description of editor commands is beyond the scope of this manual. The user is referred to the user guide for the editor (see reference below) for a description of the editor commands.

4.9.3. Temporary files

If the file being edited is larger than memory, the editor will use the two system standard temporary files, "TEMP1$" and "TEMP2$", to page the file. Each of these files must be larger than the file being edited. If these files are missing, the message "Buffer impasse." will be given and additions to the file will not be permitted.

4.9.4. For more information

Refer to the manual "Marinchip 9900 Text Editor User Guide" for descriptions of editor commands, and further information about how to call and use the editor.

4.10. FDIAG - File diagnostic

FDIAG is a program which tests I/O on a disc file, and by implication tests the disc storage that underlies the file and the operating system's file handling software. The program is invoked by a command of the form:

FDIAG ⟨file name⟩

where ⟨file name⟩ is the file to be tested. THIS FILE WILL BE OVERWRITTEN, DESTROYING ANY DATA PREVIOUSLY IN THE FILE. The user can test a specific disc unit or area by placing a file there, then calling the file diagnostic specifying that file.

4.10.1. File diagnostic operation

The file diagnostic operates by writing unique patterns in successive blocks (128 bytes) of the file until the end of file is reached. Then, the file is reset to the beginning with the SEEK$ request and the file is read back. The data in each block is validated for internal consistency, and then checked to make sure that the sector read was the expected sector. The test continues until the end of file is reached. If the test finds no errors, nothing will be printed.

4.10.2. Error messages

Cannot open named file.
: The file named on the FDIAG command could not be found in the file directory.

Write error on block ⟨number⟩.
: The operating system returned an error status on the write of the specified block. The file diagnostic terminates.

Seek error.
: The operating system returned an error status on the SEEK$ request to reset the file to the beginning. This indicates a software error in the operating system or the file diagnostic itself.

Read error on block ⟨number⟩.
: The operating system returned an error status on the read back of the specified block. The diagnostic continues

with the next block.

Bad data for block ⟨number⟩. First bad byte is ⟨number⟩. (Expected value: ⟨number⟩)

There was a data error in the block that was not detected by the operating system's disc handler. The diagnostic detected the error by internal redundancy in the block. The failing block number, first bad byte, and the expected value are printed on the error message, then the block is dumped in hexadecimal. The test continues with the next block.

Wrong block read. Expected: ⟨number⟩, received: ⟨number⟩

The block read was internally consistent, but is not the block that was written at the address that was read back. This error indicates an addressing problem in either the disc hardware or the operating system's file handler.

4.11. LINK - Linker

The Marinchip Linker is used to build an executable program from the relocatable code produced by the Assembler or the high-level language compilers. The Linker is controlled by simple commands entered from the user's terminal, and accepts its input and places its output in normal operating system files. The Linker generates straightforward English error messages for all abnormal events that occur during the process of linking. The Linker uses a virtual memory paging technique to allow itself to build programs larger than the memory available to the Linker as a work area. In fact, the Linker can produce programs larger than the memory available on the machine on which it is being run. This can be useful as programs for other users with larger memories can be generated on a minimal machine.

4.11.1. Linking a program

The linker may be used in two modes: normal mode, where commands are entered from the keyboard and the linking process is performed in an interactive mode, and shorthand mode, where all the linking information is entered on the line that invokes the linker.

4.11.1.1. Shorthand linking

In shorthand mode, the linker is called by typing the statement:

```
LINK ‹out›=[@]‹in1›,[@]‹in2›,...
```

to the operating system when at command level. "LINK" is the name of the linker, ‹out› is the name of the executable file to be created, and ‹in1›, ‹in2›, etc., are the names of the relocatable files that are to make up the executable program. If the name of an input file is preceded by an at sign (@), it is assumed to be a text file containing Linker commands (see below for descriptions of commands), rather than a relocatable file. If the input files named satisfy all external references, the executable file will be created and the linker will terminate normally. If undefined symbols remain, they will be listed, and the linker will enter normal interactive mode (see below) to allow the user to load files which define the undefined symbols.

For example, to create an executable program called "OBJ" from relocatable files named "MAIN", "CSUB1", "CSUB2", and "CSUB3", the following command would be used:

```
LINK OBJ=MAIN,CSUB1,CSUB2,CSUB3
```

4.11.1.2. Normal interactive linking

The Linker is called from the command mode of the operating system by simply typing its name, LINK. The operating system will load the Linker and execute it. When the Linker receives control, it will prompt the user for a command with a sharp sign (#).

4.11.1.2.1. Defining the output file

Once the Linker has been called, the user must specify in which file the executable file is to be placed. The OUT command is used to do this. The statement:

```
OUT ‹file name›
```

informs the Linker that the executable program is to be placed in the file ‹file name›. Only one OUT statement may be used in any call on the Linker.

4.11.1.2.2. Specifying the program base

Normally the Linker will create an executable program starting at address 0100, the standard system starting address for user programs. The user can override this assumption by supplying a BASE command before the first input file is named. The statement:

```
BASE ‹address›
```

will cause the executable program to be built starting at the specified hexadecimal ‹address› (that is, relocatable code will be loaded starting at that address). This feature is primarily of use when generating the operating system, or when writing programs intended to concurrently reside in memory. Normal user programs need not specify a BASE statement. The specified base address should be a multiple of 256 bytes (0100 hex). If the address supplied is not a multiple of 256, it will be rounded down to the preceding 256 byte boundary.

4.11.1.2.3. Naming the input file(s)

Once the output file has been specified, the user should specify all the programs that are to be linked together to make up the executable program. This will always include the main program created by the Assembler or compiler, and will frequently include other separately assembled or compiled subprograms, or programs from the system library. The files containing the relocatable object code for these programs should be named on one or more IN commands. The statement:

```
IN ‹file name›,‹file name›,...
```

will link the named files into the executable program. One or more ‹file name›s may be specified on the IN statement, and any number of IN statements may be used.

The references between separately compiled programs are made by means of external and entry symbols. These symbols are identified by six character names in both the program defining them and any programs referencing them. As programs are built into the final executable program, the Linker matches up these symbols and resolves the references to them. If after the execution of an IN statement there are references still undefined, the Linker will prompt the user for the next command with a minus sign (-) instead of the normal sharp sign (#). The user can then, if desired, list the still-undefined symbols by using the REF command (see below). If the main program is IN'd first, the linking process is complete when the - prompt goes away, since all references will have been satisfied.

The IN statement may also be used to cause the Linker to process a set of commands stored in a file. If a file name on an IN command is preceded by an at sign (@), then the commands from that file will be read and processed as if they were entered directly from the keyboard. Any Linker command may be used in a command file, and command files may be nested limited only by the system's restriction on concurrently open files and the amount of available memory. For example, if the file NEWPROG.LNK contains the commands to link a program, it would be invoked by:

```
IN @NEWPROG.LNK
```

4.11.1.2.4. Table of contents files

The LOCATE command (which may abbreviated to LOC) specifies a file containing a table of contents of a library of subprograms. The

statement:

LOC ⟨file name⟩,⟨file name⟩,...

identifies each of the ⟨file name⟩s as a table of contents file.

Each table of contents file is a text file containing one or more lines. Each line identifies a separately assembled or compiled subprogram file and names the external symbols defined in that subprogram. A table of contents statement is of the form:

⟨subprogram file name⟩ ⟨symbol⟩,⟨symbol⟩,...

where ⟨subprogram file name⟩ is the file name of the relocatable file exactly as it would be used on an IN statement to include it in the link, and the ⟨symbol⟩s are the external symbols defined in that subprogram.

When the linker reaches the end of a link (indicated by the END statement, see below), if there are any undefined external references, it will search the table of contents file entries in the order they were specified on the LOC statements, and attempt to resolve the undefined symbols. If the inclusion of a file based on its appearance in a LOC list results in the appearance of a new undefined symbol, the LOC list will be searched again in an attempt to resolve it. This process will continue until either all external symbols have been resolved, or a search of the LOC list fails to resolve any outstanding symbols, in which case the Linker will abandon the search.

When performing an interactive link, it is frequently desired to see if the LOC files specified will resolve the undefined symbols outstanding at some point in the link. The FETCH command, which is simply the statement:

FETCH

will cause the LOC list search to be performed exactly as it is done at the end of the link, but the Linker will not terminate at the end of the FETCH.

4.11.1.2.5. Listing the memory map

At the end of the linking process, the memory map may be listed by entering the statement:

MAP [+[⟨title⟩]]

This will type one line for each program loaded. The program name, defined via the IDT assembly directive, or by a specification in the compilation, will be listed followed by the address at which that program starts and the last address occupied by that program. This MAP is useful in program debugging, since it permits turning absolute addresses in the linked program back into relative addresses in the programs that made it up.

If nothing follows the MAP command, the memory map will be typed on the user's terminal. If a plus sign (+) follows the MAP command, the map will be printed on the standard printer, PRINT.DEV. If the plus sign is used, it may be followed by a title to be printed on the printer before the memory map is listed.

4.11.1.2.6. Closing out the program

After all the files that make up the program have been loaded by naming them on IN commands, all that remains is to tell the Linker to write the executable program into the output file. This is done by the statement:

```
END
```

If there are any unresolved external symbols at the time the END statement is entered, they will be listed following the warning message "Undefined symbols:". The presence of undefined symbols will not prevent the output program from being generated, but will cause it to error if any of the symbols are referenced during execution. After the Linker has written the executable program to the output file, it will exit to the operating system.

4.11.1.3. Comments

Comments may be included in the input to the Linker as lines which contain a period in column 1. Such lines are ignored by the Linker, but are useful to identify files used with the "@" feature on the IN command.

4.11.1.4. Executing the program

Programs generated by the Linker may be executed simply by typing the name of the file containing them to the operating system when it expects a command. The file containing the user program will

be loaded and executed.

4.11.1.5. If there are undefined symbols

If you have named all the files that make up your program on IN statements and are still getting the "-" prompt that indicates the Linker still has undefined symbols, the command:

```
REF
```

may be used to list them. The format of the listing will be one line for each symbol containing the text:

```
⟨symbol⟩ of ⟨program⟩
```

where ⟨symbol⟩ is the undefined symbol name and ⟨program⟩ is the name of the program that referenced it. Note that a symbol may appear in more than one message if it is referenced by more than one program.

4.11.2. Sample Linker use

The following presents an annotated example of using the Linker to construct a program. Let us suppose the user's main program object code has been put in the file MAIN by a compiler, and that subprograms SUB1, SUB2, and SUB3 are used by the program in MAIN. The object code for these three subroutines are in the files CSUB1, CSUB2, and CSUB3 respectively.

.LINK	The user loads the linker from the operating system command level.
#OUT OBJ	The Linker prompts the user with a sharp sign, and the user names the output file OBJ to hold the generated program.
#IN MAIN	The prompt reappears, and the user uses the IN command to name the main program.
-IN CSUB1	The user gets the "-" prompt indicating that more files are needed. The file CSUB1 is named.
-REF	The user decides to list

```
                                        undefineds.
SUB2 of MAIN                            The Linker lists them
SUB3 of MAIN
SUB3 of SUB1                            Note that SUB1 also
                                        references SUB3.
-IN CSUB2,CSUB3                         The user names the rest of
                                        the required files.
#MAP                                    The linker is happy and
                                        returns to the sharp sign.
                                        The user requests a memory
                                        map.
MAIN    0100-02CD                       The map is typed out
SUB1    02CE-030B
SUB2    030C-03FB
SUB3    03FC-0511
#END                                    The user asks to end
                                        linking
.OBJ                                    And calls his program
Enter the first data point:             The user's program is in
                                        control
```

4.11.3. Linker error messages

The following error messages can be generated by the Linker; each is explained. When there is a common user error that causes this error, it will be mentioned.

Bad character "‹char›" as item type.

> The character ‹char› was found in the object code file and is not valid. This normally occurs when the file you mention on an IN statement is not an object file created by the Assembler or a compiler.

Bad character in number field.

> A bad character was found in a numeric field of object code. Suspect clobbered object code file or trying to load non-object code.

Input file I/O error.

> Error reading file on IN statement. Was it properly created by the Assembler or compiler? This can also result from a disc hardware error.

Duplicate starting address in program ‹prog› ignored.

The program ⟨prog⟩, which was just named on an IN statement is a main program with a starting address, but another main program has already been loaded. The first starting address will be used for the program being linked.

Duplicate definition of ⟨symbol⟩ ignored in ⟨prog⟩.

The program ⟨prog⟩ defines symbol ⟨symbol⟩, but this symbol has already been defined in another program previously named in an IN statement. The second definition is ignored.

Absolute origin of ⟨addr⟩ in ⟨prog⟩ is below base of ⟨base⟩: ignored.

The program ⟨prog⟩ contains absolute load information which attempts to load below the Linker's standard load base. This most often results from misuse of the AORG directive in Assembly programs.

Checksum error.

The program being loaded has a checksum error in the object code. Has it been modified?

Checksum missing from record.

The program being loaded lacks a checksum on a record. Has it been modified?

End-of-record sentinel missing.

The program being loaded has a malformed record. Has it been modified?

Internal error: origin below load base.

If you have a program that causes this error, Marinchip Systems would like very much to see it.

Error swapping out page.
Error swapping in page.

These messages are caused either because the file named on the OUT statement was too small to hold the program being linked, or by a hardware error on the output file.

Bad input file specification.

The file named on the IN statement does not exist, or the file name is not well formed.

Bad output file specification.

The file named on the OUT statement cannot be created, or the file name is not well formed.

Output file already specified.

An OUT statement was entered, but an output file was already defined by a previous OUT statement. The second OUT statement is ignored.

No output file specified.

An IN statement has been entered, but no OUT statement has been entered yet. The IN statement is ignored.

Cannot open entry table. Processing continues.
The LOC specified ‹file name› cannot be found.

Entry table file input error.

The LOC specified ‹file name› could not be read in.

Insufficient table space.

The LOC-generated cross reference list of symbols and the files in which they were defined overflowed available memory space. Use a shorter list, or IN the required files explicitly instead of using LOC.

4.12. PASCAL - Sequential Pascal compiler

Marinchip Pascal is based on the Sequential Pascal compiler developed by Per Brinch Hansen for the PDP 11/45 at Caltech. Marinchip Systems has converted the compiler to run on the M9900 CPU and has interfaced it to use the I/O facilities of the Disc Executive, permitting it to interchange files with all other Marinchip software.

4.12.1. Calling the compiler

The Pascal compiler is called with a command of the form:

PASCAL(⟨source⟩,⟨listing⟩,⟨object⟩)

where ⟨source⟩ is the Pascal source program to be compiled, ⟨listing⟩ is the disc or device file where the compiler listing is to be sent, and ⟨object⟩ is the file in which the object code generated by the compiler is to be stored.

4.12.2. Executing the program

An ⟨object⟩ file produced by the compiler is executed simply by typing its name. No linking process is required.

4.12.3. Temporary files

The Pascal compiler requires that the files "TEMP1$" and "TEMP2$" be present on the system disc. If these files are not present, the compilation will abort with an error message.

4.12.4. For more information

See the manual "Marinchip 9900 Pascal User Guide" for more information on the Pascal compiler.

4.13. PACK - Compress files on disc

The Disc Executive allocates and stores files contiguously on discs. The file allocation process attempts to maximise the contiguous space available, but the process of file creation and deletion may result in the space on a disc becoming fragmented so that even though there is enough free space to create a new file, no single block is large enough to hold it. The PACK utility compresses the files on a disc and collects all the free space together into one block. After running PACK on a disc, a file may created on it with a size equal to the total free space available.

4.13.1. Using PACK

PACK is called with a command of the form:

```
PACK [*]⟨unit⟩/
```

where ⟨unit⟩ is the disc unit where the disc to be PACKed is mounted. If no leading asterisk is specified, a "safe pack" will be done. If the leading asterisk is supplied, a "fast pack" will be performed. The difference between a safe and fast pack is explained in the following section. The following are examples of PACK commands:

```
PACK 2/
PACK *2/
```

4.13.2. Error recovery in PACK

During the course of PACKing a disc, it is possible that all directory items may be changed and all files on the disc moved to new locations. This massive transformation on a disc makes the impact of an I/O error potentially catastrophic. PACK makes every effort to avoid errors and to minimise their effects.

If PACK is called with no leading asterisk, a "safe pack" will be done. In this form of PACK, after moving each file, the directory will be updated to reflect the changes. Normally, an I/O error during a safe pack will destroy at most only the one file being copied, and then only if its new location overlaps its old location.

If PACK is called with a leading asterisk, all files will be moved, and then the directory will be written out at the end. An

I/O error during a fast pack will usually totally destroy the contents of the disc.

In the case of error, PACK always analyses the damage done and reports it to the user. Because any PACK may result in the loss of some data if an I/O error happens, we urge you to first back up a disc using the CD command in DU before performing a PACK. If a backup is first made, then you may do a fast pack with impunity, knowing that if the PACK destroys the disc, you can always go back and recopy your backup and try again.

4.14. PREP - Initialise directory on unit

Before files may be created on a new disc, a Disc Executive file directory must be created first. Only once the file directory is present may CREATE be used to allocate files on the disc. PREP is used to create that file directory. PREP is called by the command:

PREP ⟨unit⟩/[,⟨specification⟩...]

where ⟨unit⟩ is the disc unit containing the new disc to be PREPed. If no ⟨specification⟩s are given, the disc will be set up as a normal single density, single sided disc. The directory will be allocated to hold up to 140 files.

When using double density or double sided discs, ⟨specification⟩s are used to inform PREP of the modes in which the discs will be used. Note that PREP simply writes out the Disc Executive directory, and assumes that the disc it is processing has been previously formatted for access. While single density, single sided discs may be used right out of the box without formatting, is is usually necessary to format discs to be used in double density or double sided modes. Refer to the documentation for the FORMAT program for the disc system you are using for information on disc formatting.

The specification "DS" causes the disc to be marked double sided. The file storage capacity and directory capacity are doubled. Note that in a system with double sided drives, a disc MUST be prepped with the "DS" specification if it is physically a double sided disc (as indicated by the index hole placement).

The specification "DD" causes the disc to be marked double density. The file storage capacity and directory capacity are doubled. A disc may be both double density and double sided. In this case the directory and file storage capacity will be four times that of a normal single density, single sided disc.

If a number appears as a ⟨specification⟩, the file directory on the disc will be allocated to provide space for the specified number of files. This form of specification may be used to override PREP's assumed directory sizes for special applications.

Examples of PREP commands are:

```
PREP 2/              - Single density, single sided
PREP 2/,DD           - Double density, single sided
PREP 2/,DS           - Single density, double sided
PREP 2/,DS,DD        - Double density, double sided
```

PREP 2/,DD,500 - Double density, 500 file directory

Before writing a new directory on a disc, PREP examines the disc to see if a directory previously existed. If either a Disc Executive or Network Operating System directory is present on the disc, PREP will ask the user:

Really want to destroy data on disc ⟨unit⟩?

which the user must answer "YES" before PREP will write the new directory to the disc. This query protects against inadvertant destruction of data by PREP, while still allowing discs to be re-PREPped without being reformatted.

4.15. RENAME - Rename file

The RENAME utility permits you to change the name of a disc file without affecting the information stored in the file. The call:

```
RENAME ⟨new name⟩=⟨old name⟩
```

will change the name of disc file ⟨old name⟩ to ⟨new name⟩. If one name is of the form "⟨unit⟩/⟨filename⟩", both names must be of that form, and the ⟨unit⟩s must agree. If no file with ⟨old name⟩ can be found, or a file with ⟨new name⟩ already exists, an error message will be given, and no change will occur.

For example, to rename file GARBAGE.TXT to be called BACKUP.WRD, one would use:

```
RENAME BACKUP.WRD=GARBAGE.TXT
```

4.16. ROMPGM - PROM programming utility

The Marinchip PROM Programming Utility allows 2708 PROMs to be programmed using the Cromemco Bytesaver PROM Programmer. The PROM Programming Utility programs the PROM, then verifies that the data has been correctly stored.

4.16.1. Programming PROMs

4.16.1.1. Erasing the PROM

The PROM to be programmed should first be completely erased by exposing it to a short-wave ultraviolet lamp. Make sure that the PROM is completely erased, as an incompletely erased PROM may surprise you with random data drop-outs at elevated temperature or after a period of time. Follow the exposure recommendations for the eraser you are using, and don't short cut the erase time, NOT EVEN ONCE. This paragraph is written from cruel experience. The experience that PROMpted this warning is one that cannot be recommended to any other sentient being.

4.16.1.2. Verifying the PROM is erased

An erased 2708 PROM will have all bits set. To verify that the PROM is completely erased, insert it in one of the Bytesaver sockets, then bring up the system and enter the command:

```
ROMPGM E=<slot>
```

where <slot> is the socket number in the Bytesaver where the PROM was placed. If the PROM is properly erased, nothing will be printed. If unerased words remain, their address and contents will be dumped. When a PROM fails erasure verification, insert it back in the eraser and try again.

4.16.1.3. Programming the PROM

The data to be placed in the PROM should be loaded into RAM. Make sure that the PROM programming utility does not overlay the data you are placing in PROM. The "Program power" switch on the

Bytesaver should be turned "ON", then the command to invoke the PROM programming utility should be entered. From operating system command level, this is:

```
   ROMPGM P=⟨slot⟩,⟨start addr⟩
or ROMPGM PE=⟨slot⟩,⟨start addr⟩
or ROMPGM PO=⟨slot⟩,⟨start addr⟩
```

where ⟨slot⟩ is the socket number in the Bytesaver where the PROM to be programmed has been placed (the sockets are numbered zero to seven, right to left, and the numbers are below the sockets on the board), and ⟨start addr⟩ is the address in RAM where the data to be programmed into the PROM starts. The ⟨start addr⟩ must be an even word address. The process of programming takes about two minutes. After programming is complete, the PROM is read back and compared with the data in RAM. If the data matches, the PROM Programming Utility simply exits to the operating system. If errors are found, a line will be printed for each word that failed to compare. The error message is as follows:

```
   ⟨RAM addr⟩: ⟨RAM data⟩  ⟨PROM addr⟩: ⟨PROM data⟩
```

where ⟨RAM addr⟩ and ⟨RAM data⟩ are address and data from RAM, and ⟨PROM addr⟩ and ⟨PROM data⟩ are the address and data in PROM. If the error or errors seem to be the failure of a few bits to set to zero, try programming the PROM again. Some less-than-prime PROMs take more than the recommended number of programming passes to program all bits. If the PROM data is all FFFF, make sure the "Program power" switch is on.

If the PE= or PO= forms of the program command are used, the data to be written into the PROM will be taken from the even or odd bytes, respectively, of the 2K area starting at ⟨start addr⟩. In other words, the command:

```
   ROMPGM PE=0,4500
```

will write the contents of byte 4500 into the first byte of the PROM, the contents of byte 4502 into the second byte of the PROM, the contents of byte 4506 into the third byte of the PROM, etc. If the PO= command had been used, bytes 4501, 4503, and 4507 would have been copied to the PROM. The PE and PO commands are useful when a program that has been developed in RAM is to be placed in 16 bit wide PROM, where all the even bytes reside in one PROM and the odd bytes in another PROM.

4.16.1.4. Turning off Program power

When the PROMs have been programmed, make sure you turn off the "Program power" switch on the Bytesaver. FAILURE TO DO THIS WILL CERTAINLY LEAD TO ZAPPING ALL THE PROMS IN THE BYTESAVER. Since systems using the Bytesaver normally use it to hold the Debug Monitor and Disc Boot PROMs, this means that the system will be down until those PROMs can be reprogrammed. Marinchip Systems can furnish, on request, information on how to modify the Bytesaver to prevent destruction of PROMs in selected sockets.

4.16.2. Verification of existing PROMs

ROMPGM can also be used to verify a PROM against data in RAM without first programming the PROM. If called:

```
   ROMPGM V=<slot>,<start addr>
or ROMPGM VE=<slot>,<start addr>
or ROMPGM VO=<slot>,<start addr>
```

the verification of data in the PROM in the designated <slot> will be carried out against the data at <start addr>, and any discrepancies will be listed in the format explained in the section "Programming the PROM" above. Since ROMPGM automatically verifies data following programming, this feature is primarily useful for verifying existing PROMs against a master data file. If the PE= or PO= commands were used in creating the PROM (see "Programming the PROM" above), the VE= or VO= command must be used when verifying programming. It will cause the data for verification to be loaded in the same manner as the data used in programming the PROM.

4.17. SIZE - Determine space required for file

The SIZE utility program examines the contents of a file and calculates the number of sectors required to hold the text found in the file. SIZE works on text files, relocatable files, and executable programs produced by LINK. SIZE is invoked by the command:

SIZE ‹file name›

where ‹file name› is the name of the file whose size is to be calculated. Note the distinction between the file size printed by SIZE and the size printed by DIRECT: DIRECT prints the number of sectors allocated to the file, while SIZE prints the number of sectors actually used by the contents of the file.

4.18. TCOPY - Text file copy utility

The Text Copy Utility is a very simple utility program provided by Marinchip Systems for the 9900 computer system. Its usefulness transcends its simple function of moving a text file from one location to another because of the generality of the file system that underlies the program. Since all peripheral devices are treated as files by the Marinchip operating systems, the Text Copy Utility can be used for functions as diverse as the following:

- Copying a disc file from one disc to another.
- Concatenating several files into one large file.
- Listing a disc file on the console.
- Making a hard copy of a listing stored on disc.

...and of course all the obvious permutations and combinations that the above immediately suggest.

4.18.1. Using TCOPY

The Text Copy Utility is invoked simply by typing the name of the file that contains it to the operating system when the operating system prompt appears. The utility is stored in the file TCOPY on a standard Marinchip system disc. Following the name of the Utility program, the destination file is specified, followed by an equal sign, and one or more source file names separated by commas:

```
TCOPY ‹ofile›=‹ifile›,‹ifile›,...
```

The ‹ofile› and ‹ifile› specifications may be fully general file names, as described in the manual for the operating system being used, and may be either device files or disc files.

The action of the command will be to copy the input files into the output file, from left to right as specified on the command. The result will be an output file consisting of all the lines in the input files concatenated. Of course, if only one input file is specified, the output file will be an identical copy of the input file.

4.18.1.1. Examples of use

To list the contents of the file MYPROG on the console:

```
TCOPY CONS.DEV=MYPROG
```

To concatenate the files PROG1, SUB1, and SUB2 into the file BIGGIE:

```
TCOPY BIGGIE=PROG1,SUB1,SUB2
```

To send the file USRDOC to the printer:

```
TCOPY PRINT.DEV=USRDOC
```

To read a paper tape into the file STUFF:

```
TCOPY STUFF=PTR.DEV
```

4.18.1.2. Error messages

The following are a list of error messages that may be generated by the Text Copy Utility and their causes:

Error: Specify ⟨ofile⟩=⟨ifile⟩,⟨ifile⟩,⟨ifile⟩,...

This message appears whenever a syntax error is detected in the specifications. Probably one of the file names is badly formed, or a delimiter between file names is incorrect.

Error reading file ⟨ifile⟩.

An I/O error was encountered reading from the named input file. The output file is closed, and any files following the named files are ignored.

Error writing output file.

An I/O error was encountered writing the output file. The Utility immediately terminates.

File ⟨file⟩ does not exist.

The named file could not be opened. If this is the output file, the command is totally ignored. If an input file, the output file is closed, and any input files following the named file are ignored.

4.19. WORD - Word processor

The Marinchip Word Processor (WORD) is a powerful yet easy to use text formatting language. It contains a set of basic commands sufficient for most text formatting applications, and provides a comprehensive string and macro facility so that the basic language may be extended by the user for more complex formatting tasks. Facilities built into WORD include:

- Right justification, centering
- Automatic reformatting for different output devices
- Multiple column output
- Automatic assignment of page and section numbers
- Automatic generation of Table of Contents

4.19.1. Using WORD

The input file for WORD is prepared using the Text Editor, then WORD is called to format the text:

```
WORD ⟨output file⟩=⟨input file⟩
```

where ⟨input file⟩ is the file containg the text to be formatted, and ⟨output file⟩ is the disc or device file where the formatted text will be placed.

4.19.2. For more information

Refer to the manual "Marinchip 9900 Word Processor User Guide" for information on how to prepare text for WORD, and further information on how WORD is used.

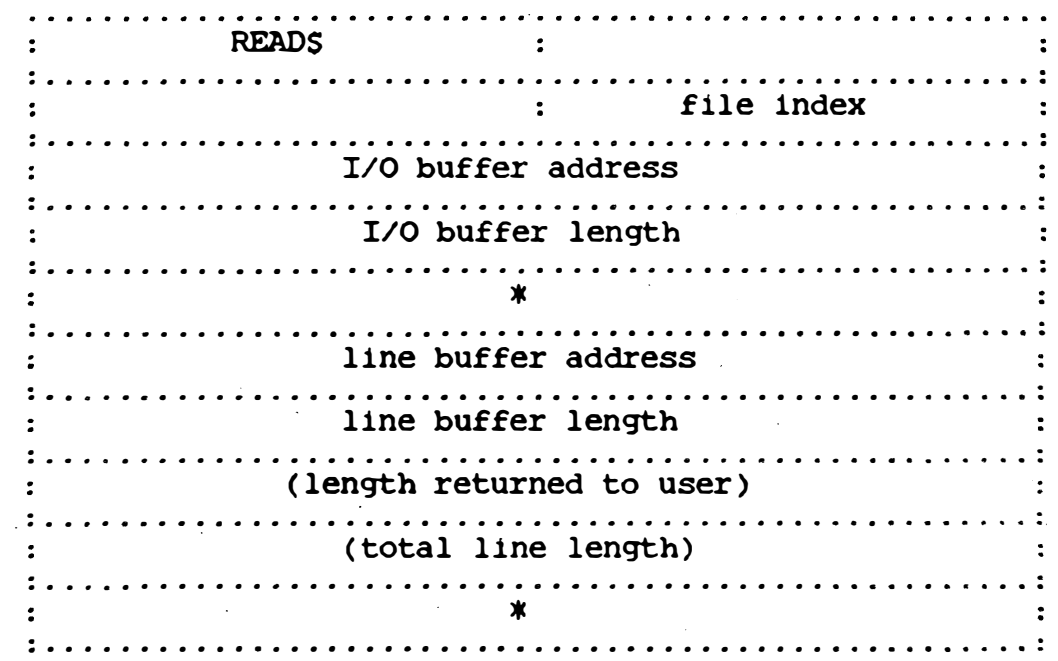
5. System library subroutines

The system disc supplied by Marinchip Systems contains a number of relocatable subroutines intended for use in user programs. These routines are described by the sections below. The sections are listed by the name of the file containing the subroutine on the system disc. The entry points and calling sequence for each routine are discussed in the description of the file.

5.1. TEXTIN.REL - Read text input file

Entry points: TEXTIO, TEXTIN

This routine is a general subroutine which reads system standard text files and returns individual lines to the calling program. All communication with the subroutine is through a packet with the following format:



The packet must be initialised with the READ$ function code, the ⟨file index⟩ of the file to be read, the address of an I/O buffer to be used to read the file ⟨I/O buffer address⟩, and its length ⟨I/O buffer length⟩. The longer the I/O buffer, the more efficient the access to the file will be. If the program is to run under the Disc Executive, the I/O buffer must be a multiple of 128 bytes. There are no restrictions under the Network Operating System. Once the above fields have been set up, the text input routine is initialised by the call:

```
LI      R1,⟨packet⟩
BL      TEXTIO
⟨return⟩
```

where ⟨packet⟩ is the address of the above packet and ⟨return⟩ is the return point following the call.

To read a line from the file, store the address of the buffer

where the line is to be read into ⟨line buffer address⟩, and set the length of the line buffer into ⟨line buffer length⟩, then use the call:

```
LI      R1,⟨packet⟩
BL      TEXTIN
DATA    ⟨I/O error⟩
DATA    ⟨end of file⟩
⟨return⟩
```

If an I/O error or end of file is encountered, TEXTIN will jump to the respective address specified following the call. If the line is read normally, control will return following the two DATA words. The ⟨(length returned to user)⟩ field will be filled with the length of the line stored in the user buffer. This value may be shorter than the user buffer, but will never be longer. The line stored in the buffer consists of just the text; the trailing carriage return is not stored. The ⟨(total line length)⟩ field is filled with the total length of the line just read, and will differ from the ⟨(length returned to user)⟩ only when the line was truncated to fit into the user buffer.

The TEXTIN routine is automatically closed out when the end of file is encountered. No special close call is required.

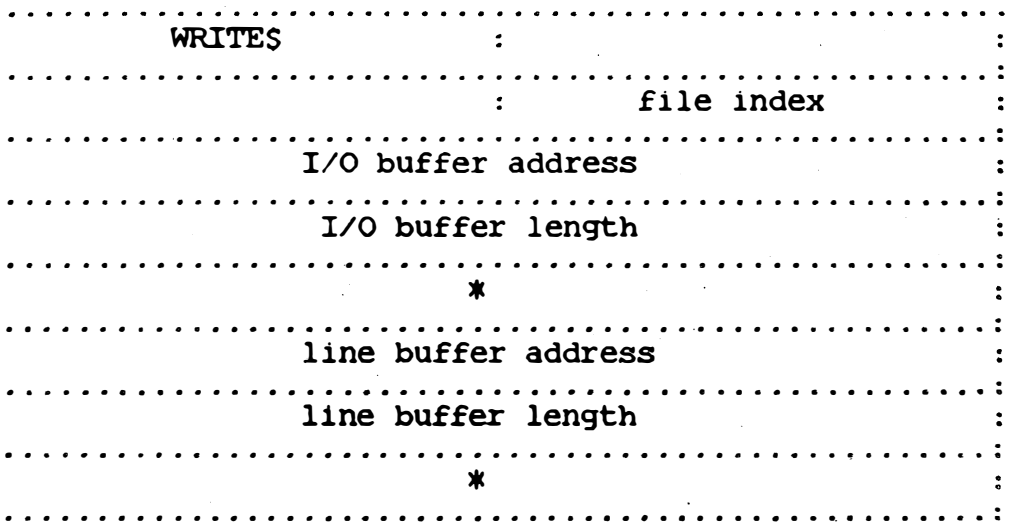
TEXTIN is completely reentrant, and may be used to read any number of text files concurrently (using one packet for each file, of course).

The fields in the packet labeled with an asterisk (*) are used by the TEXTIN routine for its own local storage. They must be provided in the packet, but need not be initialised nor examined by the user.

5.2. TEXTOUT.REL - Write text output file

Entry points: TEXTOO, TEXTOUT, TEXTOC

The TEXTOUT subroutine creates a system standard text file from lines generated by the calling program. All communication between the caller and TEXTOUT is through a packet with the following format:



In order to use TEXTOUT to generate a text file, the user must set up the packet with the WRITE$ function code, the ⟨file index⟩ of the file to be written, and the address ⟨I/O buffer address⟩ and length ⟨I/O buffer length⟩ of the buffer to be used to hold data to be sent to the file. For programs which are to run under the Disc Executive, the I/O buffer must be a multiple of 128 bytes. The Network Operating System imposes no restriction on the length of the buffer, although under both systems the efficiency increases as the buffer is made larger. Once the packet has been initialised with the above values, the following call is made to open the text output routine:

```
        LI      R1,⟨packet⟩
        BL      TEXTOO
        ⟨return⟩
```

where ⟨packet⟩ is the address of the packet, and ⟨return⟩ is the return point to the calling program. To write an output line to the file, the starting address of the line should be stored into ⟨line buffer address⟩ and the length of the line stored into ⟨line buffer length⟩, then the following call made:

```
LI          R1,<packet>
BL          TEXTOUT
DATA        <I/O error>
<return>
```

The data word following call specifies the address where TEXTOUT will jump if an I/O error occurs while writing the file. If the output is completed normally, TEXTOUT will return following that data word.

When all lines have been written to the file, text output must be closed with the call:

```
LI          R1,<packet>
BL          TEXTOC
DATA        <I/O error>
<return>
```

This call is essential, as it places the end of file mark at the end of the text file, and causes the last block of data to be written to the file.

The TEXTOUT routine is fully reentrant and may be used to write concurrently to as many files as desired (of course, one packet is used for each file).

The fields in the packet labeled with an asterisk (*) are used by TEXTOUT for local storage. They must be provided in the packet, but need not be initialised or examined by the user.

5.3. TRACE.REL - Instruction trace

Entry points: TON$, TOFF$

The instruction trace package in TRACE.REL is a powerful tool for debugging assembly language programs. The trace is activated by the call:

```
BLWP    TON$
```

Following the call, each instruction executed will be printed in assembly language format on the user terminal. Register and memory operands referenced or changed by the instruction will be edited. Conditional jump instructions will be flagged with an asterisk (*) if they actually jumped. System calls (JSYS) will be printed as if they were a single instruction (that is, the trace package will not attempt to trace into the system). The trace package will not trace itself.

The trace may be turned off by executing the call:

```
BLWP    TOFF$
```

Following return from this call, the machine will be "native mode", and will execute instructions normally.

Neither TON$ nor TOFF$ change the contents of any workspace registers or the condition code, so they may be inserted anywhere in a program.

The trace package executes instructions interpretively, so it is capable of tracing code in ROM as well as in read/write memory. Obviously, when a program is executed under the trace it executes tens of thousands of times slower than when being executed directly by the machine, so code which has to meet external timing constraints may not be able to be debugged using the trace. For most code, though, the trace should immediately show where a program is going wrong.

MDEX USER GUIDE -2

by John Walker

Marinchip Systems

Mill Valley, CA 94941