NOS/MT

MULTI-USER SYSTEM

Marinchip 9900 Network Operating System

Multi-user Version (NOS/MT) User Guide

For Release 2.5

by John Walker

Revised January 1981

Marinchip Systems 16 St. Jude Road Mill Valley, CA 94941 ■ (415) 383-1545

Marinchip 9900 Network Operating System User Guide

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

1. Introduction

The Marinchip Network Operating System is a general-purpose operating system for the Marinchip 9900 computer. The Network Operating System provides a comprehensive set of functions to programs running under its control. The general design goals of the system may be summarised as follows:

. Provide device-independent interfaces for all system calls. This enables use of new hardware as it becomes available without requiring changes in existing programs.

. Shield programs from hardware-imposed restraints, and allow for predictable expansion in hardware capacities in the coming years. The system implements a byte-addressable file space allowing individual files as large as four thousand million bytes.

. Provide a software structure allowing the configuration of multi-user, multi-tasking, multi-node networks containing an unlimited number of loosely coupled systems. The system has been designed so that the addition of these capabilities requires no redesign of the system or programs running under its control. The file privacy, user identification, and access rights required in such a system were designed into the Network Operating System from the start.

. Offer the user and application developer a "friendly" environment. The system provides powerful and convenient terminal support, flexible file naming and accessing conventions, and a complete set of software tools for software development and maintenance.

. Allow addition of services to the system without modification of the system itself. Most system services are simply user programs. The user can add to these services simply by writing new programs.

1.1. Structure of this manual

This manual begins with a discussion of general concepts underlying the design of the system. Since most system functions relate to the file system, the largest part of this section discusses the structure and use of the file system.

Next, the system is described as seen by a user at a terminal connected to the system. For most users who use high-level

languages, this will be all the knowledge of the system they require. This section discusses both the terminal support provided by the system and system commands available from the terminal.

The next section of the manual is intended for the assembly language programmer and discusses the interface to the system for programs running under its control. This section discusses both system calls and subroutines provided by the system.

The next section describes optional system features, which may or may not be present in a given system, depending on the hardware and software configuration.

Next, a summary of the standard programs provided by Marinchip Systems is provided. For some programs, the description in this section constitutes the complete documentation. For others, the appropriate Marinchip Systems user manual is referenced.

Finally, descriptions are given of all the standard relocatable subroutines provided by Marinchip Systems. These subroutines are used primarily by assembly language programmers to access various system services.

1.2. Notation conventions

The following conventions are used in examples given in this manual.

Items enclosed in corner brackets like ⟨this⟩ refer to a an item of the type described by the name within brackets. For example, ⟨number⟩ refers to any valid number.

Items enclosed in square brackets like [this] represent optional items. This notation is used both for optional repetition of items and for items for which a default is assumed when omitted. The text will explain what default is assumed when specifications are omitted.

An ellipsis (...) is used to represent optional repetition of the preceding item, separated by the preceding delimiter. When this notation is used, any number of items may be specified, limited only by the overall restriction on the length of the specification.

All examples of input to the system are given in UPPER CASE TYPE. The system is actually insensitive to the case of input, so the actual input may be either upper case, lower case, or mixed.

1.3. Implementation changes and restrictions

This manual attempts to describe the Network Operating System as accurately as possible. Due to the time required to revise and reprint a publication of this size, however, this manual does not contain information which changes from release to release of the system. That information, plus notes on known restrictions, problems, and hardware support may be found in the publication "Marinchip 9900 NOS/MT System Memorandum" which is issued with every release of the system itself.

2. General concepts

This chapter discusses the basic concepts of the Network Operating System.

2.1. The file system

The file system provided by the Network Operating System is a byte addressable, multi-level directory system. There is a separate file system on each volume (unit of storage) known to the system.

The Network Operating System offers a very general and flexible file system. Because of the large number of options and cases available, the following discussion may be difficult to understand until you have more experience with the system and the concepts on which the file system is based. Don't be upset if you don't understand all of the following information on the first reading. You will seldom have cause to use all of the features and capabilities described below, nor need you understand them to make effective use of the system for your application.

2.1.1. Storage devices

A STORAGE DEVICE is the physical equipment which is used to implement the file storage. Examples of storage devices are floppy disc drives, disc cartridge drives, and fixed disc drives. The storage device is distinct from the actual storage medium used in the device (although in the case of devices such as fixed discs, no physical distinction exists). All storage devices in the system are numbered, and referred to by their number. The lowest numbered device is 1, which is usually the device on which the system volume is mounted, and other devices are numbered 2, 3, etc.

2.1.2. Volumes, directories, and files

A VOLUME is a physical unit of storage. Examples of volumes are floppy discs, hard disc packs, and fixed disc surfaces. A volume may be REMOVABLE (such as a floppy disc), or FIXED (such as a fixed disc surface). All volumes have VOLUME NAMES, which are user-assigned names of from 1 to 14 characters which uniquely identify the volume.

Before the files on a volume can be accessed, the volume must be MOUNTED on a storage device. The process of mounting involves first physically installing the volume on the device, then issuing a MOUNT command to the system which causes it to add the volume to the list of active volumes in the system. A volume is removed from the system by issuing a DISMOUNT request to the system, and upon completion of that request, physically removing the volume. The volume from which the system was loaded will be automatically mounted by the system. Other volumes must be mounted and dismounted by the user.

Files are created on volumes. Each volume contains a table in which an entry is made for each file on the volume. This table is referred to as the FILE INDEX TABLE. The maximum number of files that may be placed on a volume is specified when the system is generated. The storage available on a volume is assigned to files either automatically as they are written into, or explicitly when a contiguous file is allocated.

A file may be either a DATA FILE or a DIRECTORY FILE. A data file is the normal kind of file used by programs, and may contain programs or data. The system imposes no structure on these files, although the programs using them may require certain data to be present. A directory file contains FILE NAMES and entry numbers in the file index table, and is used by the system to look up files on the volume. Directory files may be read by the user, but may be written only by the Network Operating System. There may be any number of directories on a volume. System calls allow the user to create and delete directories as well as regular data files.

Each volume contains at least one file, called the ROOT DIRECTORY. The ROOT DIRECTORY contains at least one entry, for itself. The file name in this entry is ".", which is the name used by a directory to refer to itself. The root directory may (and normally will) contain other names, which may be either data files, or directory files. All directories contain the name "." which refers to the directory itself. In addition, all directories other than the root directory contain the name "..", which refers to the directory in which that directory is defined. The file accessed by the name ".." is referred to as the PARENT DIRECTORY of a directory. It is possible to trace back from any directory to the root directory by continuing to access the ".." name until a directory is found which does not contain it. That directory is the root directory on the volume.

2.1.3. File names

Files are referenced by a FILE NAME. The most general form of a file name is:

```
⟨device/volume⟩:[/]⟨name1⟩/⟨name2⟩/.../⟨namen⟩
```

Because the structure of a file name is potentially complex and involves many concepts basic to the operation of the file system, let us examine several cases.

First, consider the name:

```
PAYROLL:NAMEFILE
```

The name "PAYROLL" refers to the name of the volume on which the desired file resides. If this volume is not currently mounted on the system, the file name is in error and will be rejected by the system. If the volume PAYROLL is mounted, the system will search the root directory on it for the file NAMEFILE. If that file cannot be found, the file name is incorrect. If found, the process of finding the file is completed.

Frequently it is desired to collect a group of files together that bear some relationship to one another. For example, suppose we have a Payroll system, and we wish to keep a copy of files of hours worked and paychecks printed for each month. We could code the names of these files to include the year and month, but a clearer and more useful approach would be to place the files in a directory with the name of the month. To do this, we would create in the root directory a file with, say, the name MARCH. We would tell the system that the file MARCH was to be a directory. Then, we would create within that directory files called, say, HOURS and CHECKS. We could then access the paychecks for March by using the file name:

```
PAYROLL:MARCH/CHECKS
```

Upon encountering this name, the system would look up the file MARCH on volume PAYROLL as in the NAMEFILE example above. Upon finding a slash after the name, the system would make sure that the file MARCH was a directory (and reject the name if not), and proceed to search that directory for the file CHECKS. This process can be extended to as many levels as desired. For example, a name such as the following:

```
PAYROLL:1978/JULY/MFGDIV/HOURS
```

might be used to access the file containing the hours worked for

employees in the manufacturing division of a company in July of 1978.

A complete file name is sometimes referred to as a PATH NAME, since it describes the path taken to locate the file. The path starts at the root directory on a volume and specifies all the directories leading finally to the file desired. Each segment of the path name refers to a name in the directory represented by all the names to its left. These names are from 1 to 14 characters in length, and are composed of upper-case ASCII letters, numbers, and the special characters:

$. -

Names may be supplied to the system in either upper or lower case, but will be converted to upper case before use. Hence, case is insignificant in looking up files.

2.1.4. Multiple names for a file

A file must have at least one entry in a directory in order to exist, but may have as many entries in as many directories as desired. Each entry in a directory is referred to as a LINK to a file. There is nothing special about the first (or original) name of a file: all links are equivalent. There is a system call which can be used to create a new name that points to an existing file. A file will not be physically deleted as long as there are any links to it. The system call which deletes a file removes a specific name for the file (after which its name can be immediately reused for another file), but only releases the storage for the file if the name being deleted is the only name for the file. Consider the following case:

1. User Joe creates a file DSK1:UTIL/MYSTUFF
2. User Charley creates a link to Joe's file called DSK1:CHARLEY/JOESTUFF
3. Joe deletes DSK1:UTIL/MYSTUFF

At this point, the directory entry for Joe's file is deleted, but since Charley still has a link to the file, the file itself is not deleted. Joe is free to create a new file called DSK1:UTIL/MYSTUFF, and that file will be distinct from the file Charley accesses as DSK1:CHARLEY/JOESTUFF. When Charley deletes DSK1:CHARLEY/JOESTUFF, the file will be physically deleted and its space released, since that deletion removes the last link to the file.

Only data files may have multiple names. The system does not

allow multiple names for directories. This restriction is enforced to make the design of programs which process the file directory more straightforward, not because of any structural reason.

2.1.5. Assumed volume and directory

In normal use of the system, users will do most of their work within one directory, and on one volume. To eliminate the need to specify the entire name of a file (which may contain many names separated by slashes), the system allows the specification of an assumed volume and directory. When the system encounters a file name with no volume specification and no leading slash, the search for the file will commence in the assumed directory on the assumed volume. For example, let us assume we have a system where each department using the system has its own directory, within which there is a directory for each user. If Joe Smudley works for the Engineering department, his directory might have a name like:

```
PACK14:UFILES/ENGINEERING/SMUDLEY
```

Let us assume that Joe works on several projects, and has grouped the files for each project in a directory within his directory. A typical data file name, then, might be:

```
PACK14:UFILES/ENGINEERING/SMUDLEY/256KRAM/MASK
```

where 256KRAM is the directory Joe has created for files relating to that project, and MASK is a file within that directory. Now unless Joe is a very fast typist and has a very durable keyboard, he is likely to complain that the system is somewhat cumbersome to use if he has to specify that name every time. The system can be set up to assume any portion of the file name, starting at the volume, and extending as far as desired. In addition, the assumption can be made automatically for each user at the time he signs on to the system. The system Joe uses will probably have been told to assume:

```
PACK14:UFILES/ENGINEERING/SMUDLEY
```

when Joe signs on. Joe will then be able to say simply:

```
256KRAM/MASK
```

and the system will automatically get the right file. If he wishes to work entirely within the directory 256KRAM, he can tell the system to assume it, and reference his file simply as:

MASK

But, you ask, what happens if Joe wants to read one of Charley Smushbender's files? It is not necessary to undo the assumption of the directory, only to specify the complete name with a leading slash (/) before the first name. For example:

/UFILES/ENGINEERING/SMUSHBENDER/64BITMPU/PINOUT

This will assume the same volume as Charley's assumed directory, that is, PACK14. If the file desired is on another volume, Joe can specify that volume name. Whenever a volume name is specified as part of a file name, the search for the name will start in the root directory of that volume, so there is no need to specify a leading slash in such a name.

2.1.6. File privacy and access restrictions

Associated with each user of the system is a GROUP NUMBER and USER NUMBER. Both are numbers in the inclusive range from 0 to 65535, and identify which group (department, division, tribe, etc.) the user belongs to, and his number within that group. The system will set these numbers automatically from the user database when the user signs on to the system. There are three basic things you can do with a file:

Read it
Write it
Execute it (or if a directory, look up files in it)

The system allows you to specify whether these things can be done by:

Yourself
Members of your group
Everybody else

You can specify the privileges associated with a file when you create it. If you don't specify, default privileges gotten from your user database entry will be used. You can change the file privileges only of files you own.

In addition to the file privacy modes, a user can specify additional access restrictions at the time a file is opened. For example, a user may be permitted to both read and write a file by the file privacy bits, but may choose to open the file for input only to protect against inadvertent destruction of data in the file. All file privacy checks are performed at the time a

specific request is made referencing the file. Hence, changes in the file privacy bits will take effect immediately.

The file privacy bits also apply to directory files, but with a slightly different meaning. The Execute privilege refers to the ability to search a directory, that is, to have the system look up files in it. The Read privilege continues to control whether a user may open the directory and read its text like any other file (note that one may not list the directory with the DIR utility without having Read permission for it). Users are not allowed to directly write directory files, as all directory maintenance is done by the system. However, the Write privilege is still meaningful for directories. A user may not make any request that would cause the system to modify the directory (such as creating a new file in it, or deleting a file in it) unless he has the Write privilege on the directory.

2.1.6.1. Privileged mode

A user logged in with group number 0 and user number 0 is considered to be PRIVILEGED. The system will bypass all file privacy and access restriction tests for requests submitted by such a user. In addition, certain system commands and system calls are available only to privileged users. These restricted commands are identified in their descriptions in this manual. The privileged user concept is necessary to bypass system security for system-wide maintenance functions (such as backing up user files), and to restrict access to functions whose unrestricted use might imperil system integrity.

2.1.6.2. Directory creation mode

A special privileged mode enables a user to create an empty file directory on a blank volume, or to recreate the file directory on a volume previously used for another purpose. Since this operation may result in the irretrievable loss of files stored on a volume, extra special security is attached to this function. The function of creating directories may only be executed if the requestor is logged into the system with group number 0 and user number 666. The password associated with this user is normally revealed only to those with a genuine need to know, and changed frequently.

2.1.7. File addressing and space allocation

The Network Operating System considers a file to be a collection of bytes. Any number of bytes may be read or written by one I/O request. Associated with each open file is an I/O pointer, which contains the number of the byte to be read or written next. The pointer starts at zero (the first byte of the file) when the file is initially opened, and is incremented by the number of bytes transferred when any data is read from or written to the file. Thus, to access the file sequentially, the user need only read or write the file. A system call is provided by which the user can set the I/O pointer to any desired value. Use of this call allows random access to the data in the file. The file may be read or written starting at any byte boundary; the system will worry about physical device block sizes and perform the correct operations. The file pointer is a 32 bit number, permitting file addresses as large as four thousand million bytes.

When a file is created, no space is initially assigned to it. As the file is written, the system automatically allocates space from the pool of free space on the volume on which the file resides and assigns it to the file. Files will never contain "holes": if the first write to a file is at byte 1,000,000, all the space from address zero to that byte will be allocated by that write. If a write requires allocation of space and there is no space left on the volume, the I/O request will be rejected by the system.

Optionally, the user may request that the system allocate a contiguous block of space for a file. Contiguous files can be random accessed faster than normal files, and are also useful for large-volume I/O applications such as program loading. A request for contiguous allocation may be rejected when enough space is available for the file because that space contains no contiguous area as large as desired. The system attempts to allocate space so as to maintain the largest possible contiguous area, but user file creation and deletion may thwart this process.

2.1.8. Memory files

The Network Operating System allows banks of memory to be configured as file storage devices. These blocks of memory may be as large or as small as desired, allowing for both high speed scratchpad usage or very large memory applications requiring speed that only memory can deliver. Memory banks configured as storage units work exactly like a fixed disc unit, except that all data stored in the memory file is, of course, lost when power is removed from the system. Otherwise, no special considerations are

required when using a memory file. Thus, programs may be made to use memory files when they are available, and disc storage when they are not. Memory files may be used in conjunction with the file and record locking mechanism (FLOCK$) to provide high speed intercommunication between programs running on the same system.

2.1.9. Device files

All system peripherals, such as the user terminal(s), printer(s), paper tape readers and punches, etc., are included in the file system. These "Device files" are given special names within the Network Operating System. These names are specified with no volume specification, and are treated as always within the current directory. Consequently, user files may not have the names of system-defined device files. The device files present in a system depend upon the system configuration. Only the user terminal device file and the parameter string device file are always present. The standard names assigned to device files are as follows:

CONS.DEV	User terminal
PARAM.DEV	Program parameter string

Other device files may be added if additional hardware support is configured in the Network Operating System. The only restrictions in use of device files stem from their hardware limitations: it is meaningless to input from a paper tape punch, or to rewind a printer. Attempts at such levity will normally be ignored by the system.

Additional device files may be added when the system is generated. An installation will normally provide device files to allow access to whatever nonstandard hardware is present at the site. In addition, the device file mechanism is used to implement various system features such as background batch and offline file printing. For information on locally implemented device files, consult the persons responsible for maintenance of the system at your installation. The device files used by system features will be discussed in the sections of this manual that describe those features.

2.1.10. 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. For some applications, such as reading and writing discs to be used on other systems or initialising discs to be used with the system, it is necessary to be able to read and write a disc directly. The Network Operating System provides this facility through the disc device file mechanism. A disc volume may be mounted for arbitrary format access through the device file mechanism by using an asterisk as the volume name on the MOUNT command. The volume will be made available for I/O through the device file, but will not be incorporated into the file system as it would be on a normal mount. Once the volume is mounted, the user may then open the device file for the unit on which the volume is mounted. This is done by opening the file named:

⟨device⟩:DISC$.DEV

where ⟨device⟩ is the name (1, 2, 3, etc.) of the disc device.

Privileged users may use the DISC$.DEV mechanism to read and write any volume in the system, while nonprivileged users may use the DISC$.DEV file only on volumes explicitly mounted for arbitrary access.

2.1.11. 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 general disc file name may be used.

2.1.12. File access procedure

Normal access to a file consists of opening the file, performing accesses to it, then closing the file. Before a file can be accessed, it must be opened. The OPEN$ request takes a file name, opens the file for access, and returns a FILE INDEX by which the user refers to the file in subsequent requests. The CREATE$ request acts like the OPEN$ request, except that the file will be created if it does not already exist (OPEN$ will error if the file does not already exist). Accessing the file is done using the various system calls to read and write the file. When the user is done with the file, it should be closed with the CLOSE$ request.

2.1.12.1. Predefined file indices

When a program receives control from the system, the file indices 0, 1, and 2 will be already opened. These indices are assigned as follows:

Index	Use
0	Standard Input
1	Standard Output
2	Message Output

Standard Input (0) is where programs normally receive their commands, Standard Output (1) is the file where normal program output is directed, and Message Output (2) is where error messages which require immediate user action should be directed. The Standard Input file is initially assigned to the user's terminal, but may be attached to another file with the SCRIPT command (or SCRIPT$ system call). The Standard Output and Message Output indices are currently always directed to the user's terminal, but later releases of the system will permit the user to redirect these files also.

3. Using the system from a terminal

This chapter describes the 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. The later chapter of this manual, "System utility programs", is also of interest to non-assembly language programmers. It provides a summary of commands which are not part of the Network Operating System itself, but are provided by Marinchip Systems on the standard system volume.

3.1. 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 will print the system sign on message on all terminals:

```
Running on MT:  Marinchip NOS/MT ver 2.5
x system(s), nnK user space.
```

The "x" in the message will be the number of systems connected together in the resource-sharing network, and the "nn" will be the size of the free space available for user programs in multiples of 1024 bytes.

3.2. Logging in

Once the system has been loaded, the user will be asked to identify himself by the message:

```
Enter user name:
```

The user should respond with his user identification name (assigned by the system manager). If the user name is incorrect, the user will be asked for the user name again. If the user name is on file with the system, the system will respond with the

query:

```
Enter password:
```

and the system will expect him to enter the correct password for the user name. The password will not be echoed to the user's terminal as it is entered. Any of the local editing keys on the terminal will cause the password entered so far to be discarded and the password prompt to be retyped. The Control C key will restart the log in process with "Enter user name:". If the password is incorrect, the "Enter password:" prompt will be reissued until the correct password is supplied (or the log in aborted with Control C).

When the proper password is supplied, the user will be granted access to the system. A message will be printed informing the user of the group and user numbers assigned to the user name he used, and any initialisation specified for the user will be performed.

The user may sign off and log back in with a different user name simply by using the command "LOGIN". Refer to the section on LOGIN in the chapter "System utility programs" for a more complete description of the use of and functions available in the log in process.

3.3. System command mode

Once the user has successfully logged in and any initialisation is complete, the system will type the system command prompt. This is simply a colon (:) typed at the start of a line. When this prompt is typed, the system is waiting for the next command from the user. At this point the system will accept a system command, or the name of a program to be executed.

3.4. Executing programs from command mode

If the name typed by the user is not a special system command (see below), it will be assumed to be the name of a file containing a program to be executed. The file name typed by the user will be looked up in the user's current working directory (see the section "Assumed volume and directory" above), and if found will be loaded and executed. If the name cannot be found in the user's directory, the system library file:

```
1:BIN
```

will be searched for the name entered by the user. If the program is found in the system binary library, it will be executed. If the program is not found in either the user directory or the system library, the system will respond with an error message and return the user to system command mode.

Programs executed from the terminal may be either executable binary programs created by the Linker (LINK), or pseudo-code files generated by compilers such as BASIC and Pascal. If the program named is pseudo-code, the system will determine this from examination of the file and automatically load the correct interpreter package to execute the pseudo-code. The user need not know what type of file is being executed. Device files may not be executed as programs.

3.5. User commands

The system contains a very small set of built-in commands. These commands are executed directly by the system rather than being executed from disc as are the commands described in the later chapter of this manual titled "System utility programs". The built-in commands are built-in because they provide functions that for various reasons cannot practically be made disc resident.

3.5.1. DISMOUNT - Dismount volume from storage unit

```
DISMOUNT <volume/device>:
```

The volume designated will be dismounted from the storage unit on which it is mounted. Note that either the device or the volume name may be specified. If any files are currently open on the volume, the command will be rejected. After the DISMOUNT command has successfully completed, the volume may be removed.

The most common problem encountered when trying to DISMOUNT a volume is forgetting that you have ASSUMEd a directory on that volume. The system will not permit the volume to be dismounted until that reference is dropped. See the discussion of the ASSUME utility program for more information on this operation.

3.5.2. MOUNT - Mount volume on storage device

```
MOUNT <device>:<volume>
MOUNT <device>:
```

MOUNT <device>:*

The MOUNT command is used to inform the system that a volume has been mounted on a physical storage device and that files on it should be made accessible. The actual mounting of the volume is done first, then the MOUNT command is issued. The first form of the MOUNT command is the most common. The <volume> name given is verified against the name on the volume mounted, and the command rejected if the names do not agree. The second form of the MOUNT command will accept any volume mounted on the <device>, as long as it is a valid file system volume. The third form of the MOUNT command is used to mount volumes which are not formatted for use with the file system. Volumes mounted in this manner may be accessed only through the disc device file (DISC$.DEV) mechanism. This type of MOUNT is used primarily when using discs used to interchange information with other systems. A volume containing system files may not be mounted for arbitrary access through the DISC$.DEV file unless the user requesting the MOUNT is privileged. This restriction prevents destruction of system files through the DISC$.DEV mechanism.

3.6. Terminal support

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

3.6.1. Terminal input

The user may type input on the terminal 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. In addition, the user may "type ahead" input before it it requested by the system or a program. There is a limit on the number of lines that may be typed ahead. When this limit is reached, the system will refuse to accept any more lines (by not responding to the RETURN key) until the system or program accepts one of the lines queued for it. This input lock situation can be escaped with the Control C key (see below). Several special functions are provided by control keys while input is being entered.

3.6.1.1. 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 terminal, 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.

3.6.1.2. Character delete

The last character typed may be deleted by pressing the Backspace (Control H) key. If the terminal 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.

3.6.1.3. 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 terminal is a CRT display, the word will physically disappear from the screen and the cursor will back up over it.

3.6.1.4. Retype input line

All input line editing with the above special keys is very easy to understand and use if the terminal is a CRT display. If the terminal is a hard copy device, 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.

3.6.1.5. 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.

3.6.1.6. 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.

3.6.2. Terminal output

Output sent to the terminal 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 terminal device. Note that control characters sent to the terminal 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 user terminal.

3.6.2.1. Output pause

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

3.6.3. Program interrupt key

An executing program may be interrupted by pressing the Control C key at any time. Any input or output in progress or queued will be aborted. If the program has requested the terminal interrupt, it will be diverted to its interrupt point so that the interrupt may be serviced. If the program does not service the terminal interrupt, it will be terminated. A program can disable the program interrupt key by assuming direct control of the user terminal. In this "raw mode", the program is responsible for handling all control characters and may take whatever action it desires when any character is pressed.

4. Using the system from a program

The Network Operating System provides three basic services to programs running under its control: a set of system calls to perform services provided by the system, information regarding parameters passed to the program by its caller and about the environment in which the program is executed, and a set of common subroutines used by most software in the system provided to reduce the size of the many programs that use them.

4.1. 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 system. 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 system. A zero status always indicates normal completion of the request. A nonzero status signifies some abnormal event. A table of status codes and their meanings is given following the description of the system calls. Any nonobvious use of status codes by specific system calls is noted in the description of the call.

The following paragraphs describe the system calls. In the paragraph heading, the mnemonic for the system call is 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.

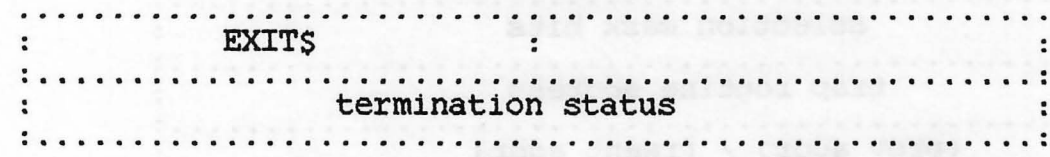
A file which defines the mnemonics for the system calls is provided by Marinchip Systems on the standard system disc. It may be included in an assembly language program by the statement:

```
COPY      "1:SOURCE/JSYS$"
```

4.1.1. Process control

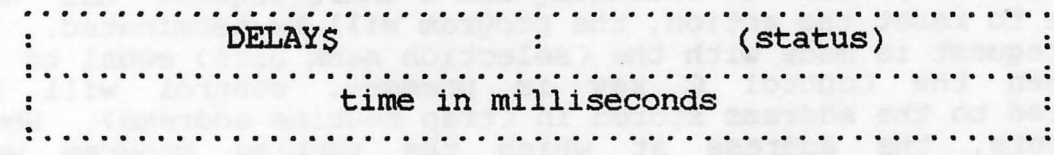
The process control requests control the processes, state, and memory allocation of an active program.

4.1.1.1. EXIT$ (02) Terminate process



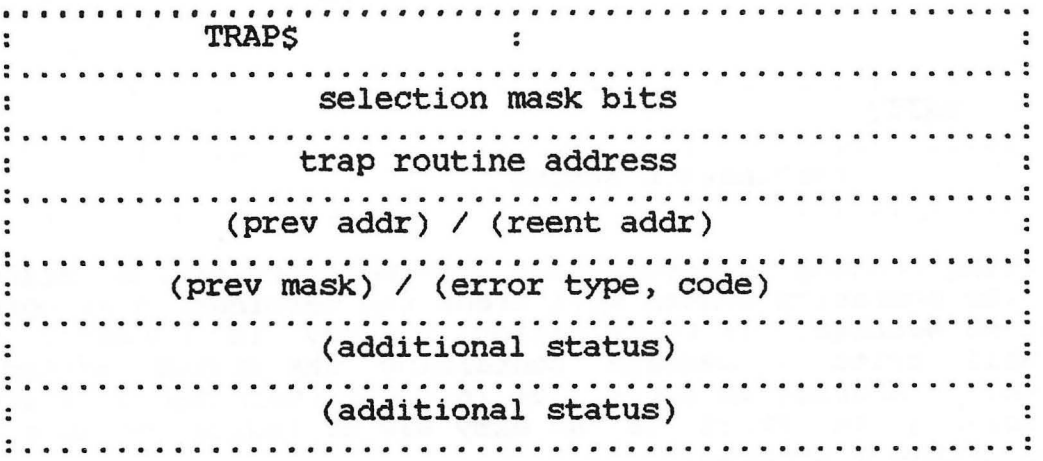
The executing process is terminated. If the ⟨termination status⟩ is zero, the operating system will treat the termination as normal and print no message. If ⟨termination status⟩ is nonzero, the system will print a message containing the status (edited in hexadecimal). Storing an error code into the ⟨termination status⟩ and performing an EXIT$ is an easy way of indicating an error condition within a program.

4.1.1.2. DELAY$ (04) Timed delay



The process executing the DELAY$ request will be delayed for a time interval approximately equal to the specified ⟨time in milliseconds⟩. Since on many configurations the resolution of the real time clock is substantially less than one millisecond, and also since other system work may delay immediate return to the delayed process, this request cannot guarantee precise timing. In the requested delay time is less than the minimum time resolution of the system real time clock, that minimum time will be the length of the delay.

4.1.1.3. TRAP$ (0E) Reset error action

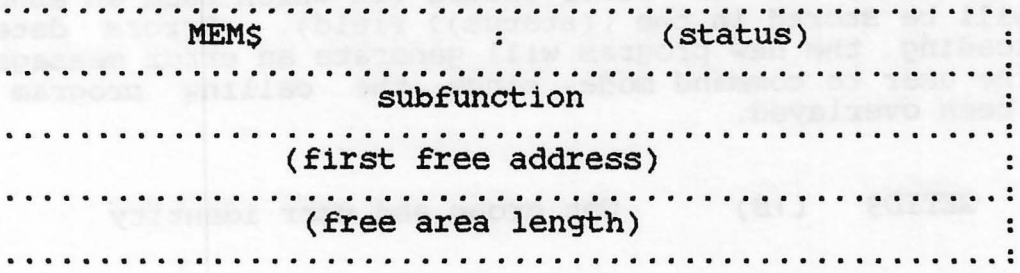


The TRAP$ request allows a program to catch various errors which may occur during execution of a program. Currently, the only error action which may be reset using the TRAP$ request is the Control C interrupt from the user terminal. If Control C is pressed while a program is executing and a TRAP$ request has not been made to reset the action, the program will be terminated. If a TRAP$ request is made with the ⟨selection mask bits⟩ equal to 1, then when the Control C key is pressed, control will be transferred to the address stored in ⟨trap routine address⟩. When this occurs, the address at which the running program was interrupted will be stored in the ⟨(reent addr)⟩ field, and the error type and code (zero for the Control C interrupt) will be stored in the ⟨(error type, code)⟩ field. If it is desired to return control to the interrupted program, it is necessary only to jump to the address stored in the ⟨(reent addr)⟩ field. The error action may be restored to the system's normal default assumption by setting the selection mask bit for that error to zero. If the entire ⟨selection mask bits⟩ field is zero, standard action will apply for all errors, and the value in ⟨trap routine address⟩ is irrelevant. Whenever a TRAP$ request is executed, the ⟨prev addr⟩ field will be set to the address of the request packet used in the last TRAP$ call made, and the ⟨prev mask⟩ field will be set to the mask bits used in that call. This information permits subroutines wishing to reset error action to return it to the state in effect when they were called. The ⟨(additional status)⟩ fields are currently unused, and are reserved for future extensions of the Network Operating System.

When writing programs which use the TRAP$ request, it is important to remember that the program may be interrupted at any time. Hence, user programs must make sure that the action taken by the interrupt does not leave the program in an invalid state. Normally, the best way to handle a TRAP$ interrupt is to set a flag and return to the interrupted program, which then tests the

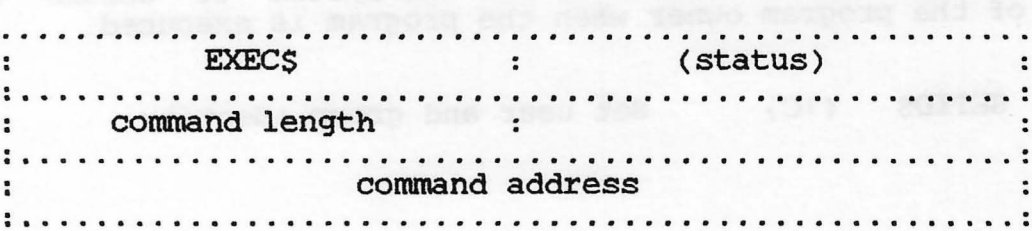
flag later and takes the desired action.

4.1.1.4. MEM$ (0F) Determine memory limits



The MEM$ request is used by a program to determine the limits of memory available to that program. To determine the free memory available, the ⟨subfunction⟩ must be set to 1. Upon return from the MEM$ request, the ⟨(first free address)⟩ field will contain the address of the first byte of storage following the current program and its parameter table, and the ⟨(free area length)⟩ field will contain the length of this area in bytes. Use of the MEM$ request allows programs to automatically adapt themselves to use however much free memory is available on the system on which they are run. MEM$ is particularly useful when used in conjunction with the dynamic buffer allocation routines described in the "System subroutines" section later in this manual. A program may determine the amount of free memory available with MEM$, construct a buffer pool consisting of that space using BEXP$, and then allocate and release storage from the pool using BGET$ and BREL$.

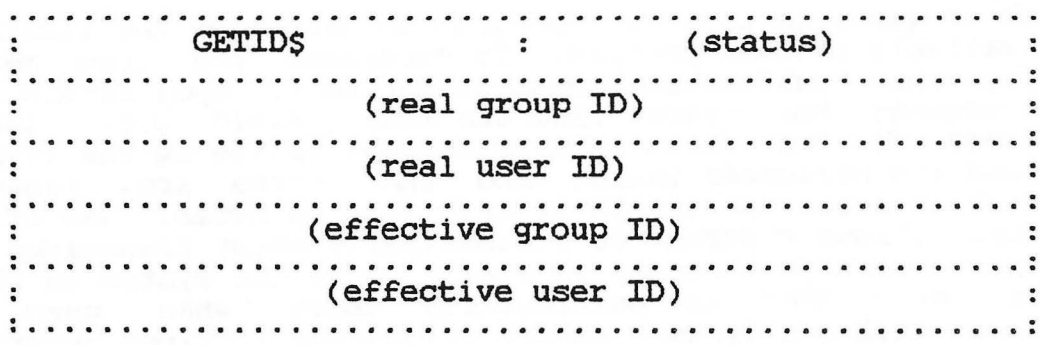
4.1.1.5. EXEC$ (11) Execute a program



The EXEC$ request permits one program to load and execute another program. The program loaded will overlay the calling program, so return to the calling program is not possible. ⟨command address⟩ is the address of an ASCII string containing the command to call the program to be executed, and ⟨command length⟩ gives the length

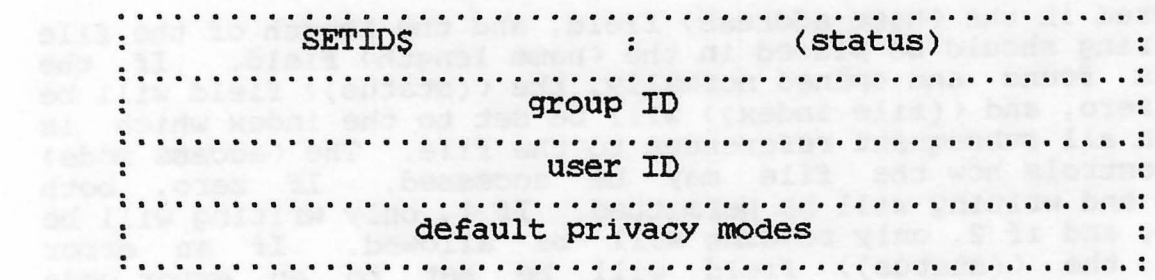
of the command string. The command string is identical to the command which would be typed on the user's terminal to call the program, and may contain parameters following the program name. Since normal completion of an EXEC$ request causes the calling program to be overlayed, control will return following the JSYS instruction only if an error occurs (in which case an abnormal status will be stored in the ⟨(status)⟩ field). Errors detected while loading the new program will generate an error message and return the user to command mode, since the calling program has already been overlayed.

4.1.1.6. GETID$ (1B) Get group and user identity



This request allows an executing program to determine the identity of its caller. The packet will be filled with the group and user numbers of the user who is logged in on the terminal (in the fields ⟨(real group ID)⟩ and ⟨(real user ID)⟩), and the group and user identification the current program is executing under (in the fields ⟨(effective group ID)⟩ and ⟨(effective user ID)⟩). The real and effective fields will differ only if the currently executing program belongs to a different user than the calling user and has the mode set which causes the system to assume the identity of the program owner when the program is executed.

4.1.1.7. SETID$ (1C) Set user and group identity

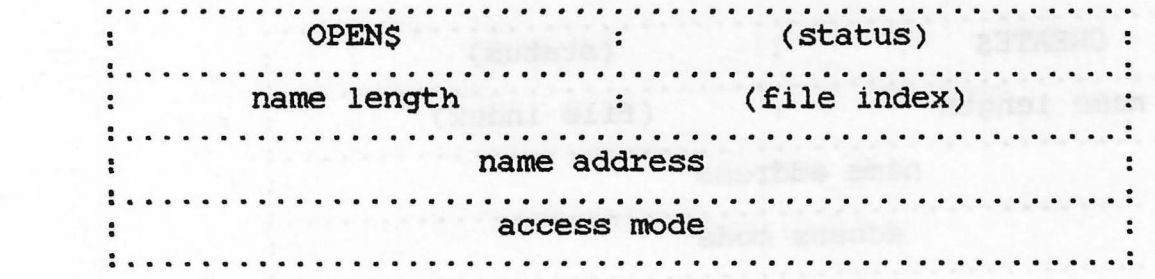


The SETID$ request sets both the real and effective identity of the user to the contents of the ⟨group ID⟩ and ⟨user ID⟩ fields. A privileged program may change the identity to any desired values, but a nonprivileged caller may only change the identity back to the real group and user identity of the user. This permits a program running under the identity of its owner to assume the identity of its caller (after having determined it via GETID$). Note, however, that such a program may not change back to the identity of its owner, as that would violate the rules for a nonprivileged call to SETID$. The ⟨default privacy modes⟩ field specifies the mode bits to be assumed for the CREATE$ request when the ⟨privacy modes⟩ field in the CREATE$ packet is zero. See the discussion of CREATE$ below for a description of privacy mode bits. If this field is zero, the default privacy modes will not be changed.

4.1.2. File control

The file control system calls include requests to create, delete, open, close, read, write, position, and to set and read various file modes.

4.1.2.1. OPEN$ (05) Open a file



The OPEN$ request opens a file for access. The address of the ASCII string containing the name of the file to be opened should

be stored in the ⟨name address⟩ field, and the length of the file name string should be placed in the ⟨name length⟩ field. If the file is found and opened normally, the ⟨(status)⟩ field will be set to zero, and ⟨(file index)⟩ will be set to the index which is used in all subsequent references to the file. The ⟨access mode⟩ field controls how the file may be accessed. If zero, both reading and writing will be permitted. If 1, only writing will be allowed, and if 2, only reading will be allowed. If an error occurs, the ⟨(status)⟩ field will be set to an error code indicating the nature of the error (see the section "System call error codes" below), and the file will not be opened. There is a limit on the number of files a program may concurrently have open. This limit is specified when the system is generated. Contact the person responsible for system generation at your installation for the limit in the system you are using. The OPEN$ request is used to open both disc and device files. All files must be opened before being used. A given file may be open more than once: a unique file index will be assigned for each OPEN$ request, and separate address pointers will be maintained for each open instance of a file.

4.1.2.2. CLOSE$ (06) Close a file

CLOSE$	(status)
	file index

The file with the specified ⟨file index⟩ (the index which was returned when the OPEN$ was done on the file) will be closed.

4.1.2.3. CREATE$ (07) Create a file

CREATE$	(status)
name length	(file index)
name address	
access mode	
privacy modes	

The CREATE$ request acts identically to the OPEN$ request, except that the named file will be created if it does not already exist. If the file does already exist, all storage assigned to it will be released. If the file does not already exist, the ⟨privacy modes⟩ field will be used to set the privacy mode bits for the file. If the ⟨privacy modes⟩ field is zero, the default privacy modes for the current user will be used.

If the file privacy is to be specified by the ⟨privacy modes⟩ field, rather than using the default for the user, the field should contain the desired permissions for the file, computed as the sum of one or more of the following permission bits:

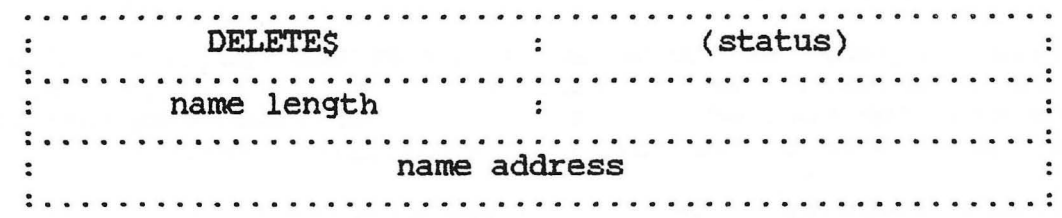
0200	Assume identity of owner when executed
0100	Read by others
0080	Write by others
0040	Execute/Search by others
0020	Read by group members
0010	Write by group members
0008	Execute/Search by group members
0004	Read by self
0002	Write by self
0001	Execute/Search by self

These bits control the basic operations of reading, writing, and executing (or searching for a directory file), independently for the user himself (group and user number of file owner equal to user's group and user number), members of his group (group number equal, user number different), and others (group number unequal). The privileges for each separate class of users are completely distinct: it is possible, although not very useful, to create a file which can be accessed only by users other than the user who created the file. The "Assume identity" bit applies to files which are executed as programs. When a file is executed which has this bit set, the program will execute under the group and user identity of the file owner rather than the identity of the user who called the program. The GETID$ and SETID$ calls may be used to determine and change this status. This feature is primarily used by programs which selectively access restricted data for a user. The program is granted access to a file which the user cannot directly access, and is responsible for selecting and delivering to the user data from this file relevant to the user.

Note that since CREATE$ releases storage assigned to a previously existing file, performing a CREATE$ on a contiguous file whose space was previously assigned via ALLOC$ will cause that space to be released, and hence the file to revert to non-contiguous allocation. As a result, programs which write into contiguous files should be careful to open them with OPEN$ rather than CREATE$, which would cause release of the previously allocated

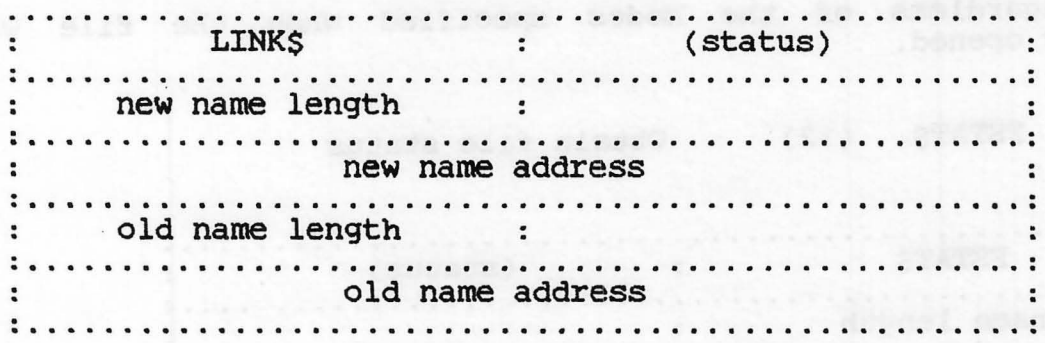
contiguous block.

4.1.2.4. DELETE$ (09) Delete a file



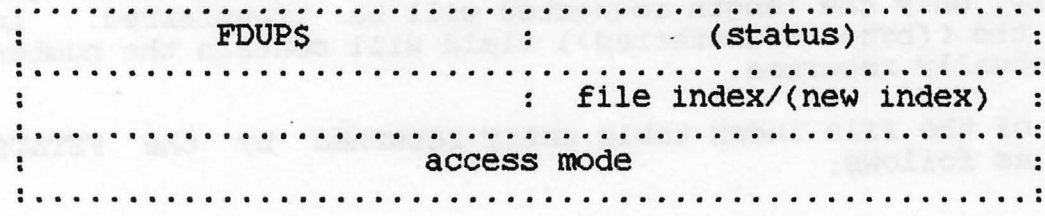
The address of the name of the file to be deleted is stored in the ⟨name address⟩ field, and the length of the name is stored in the ⟨name length⟩ field. The DELETE$ request will delete the name from the file directory. If the name deleted is the last name associated with a file, the space assigned to the file will be returned to the pool of free space on the volume on which it resides. If the file being deleted is a directory, the system will check whether the directory contains any file names. If the directory is void (contains only its self pointer "." and the parent pointer ".."), the deletion will be allowed. If the directory contains any other names, the deletion will be rejected with a status of 16, which signifies an operation which requires a void file was not given one. The DELETE$ function may not, then, be used to delete a directory until all names in the directory have first been deleted using DELETE$. If the file being deleted via DELETE$ is open, the directory entry for the file will be deleted, but if that directory entry is the last name for the file, the space will not immediately be released. Rather, a "drop flag" will be set, so that when the last open instance of the file is closed the space assigned to the file will be released to free space. Thus, a program is assured that once a DELETE$ is completed it may immediately create a new file by that name, but other open references to the file may continue to reference it.

4.1.2.5. LINK$ (08) Create link to file (alias)



The LINK$ request creates a new name which designates the same file as an existing name. The address of the existing file name is stored in ⟨old name address⟩ and its length is stored in ⟨old name length⟩. The address of the new name is stored in ⟨new name address⟩ and its length in ⟨new name length⟩. If the request completes normally ((⟨status⟩) is zero), the new name will access the same file as the old name. The request will be rejected with a bad status if the old file name cannot be found, if the new file name is already in use, or if an attempt is made to create a link from one volume to another. In addition, nonprivileged users are not allowed to create additional names for directory files. If the ⟨device/volume⟩: specification is omitted on the new name, it will be assumed to be the same as that used for the old name. LINK$ may not be used to create additional names for device files.

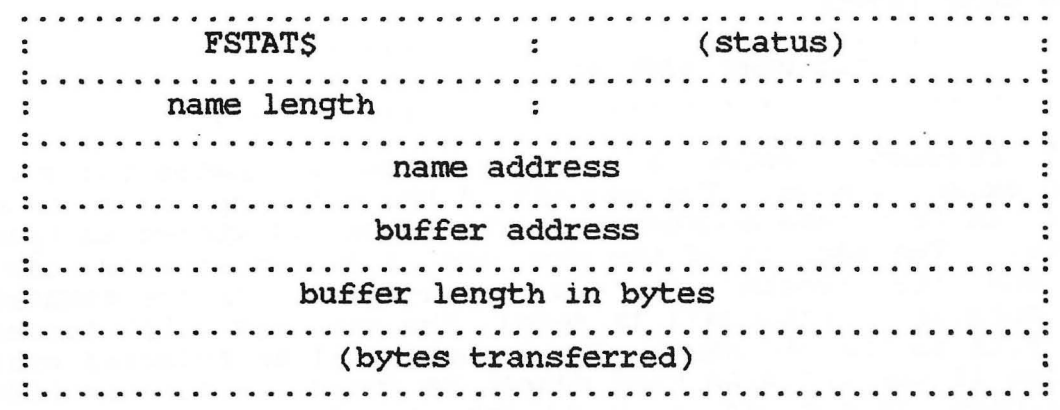
4.1.2.6. FDUP$ (22) Duplicate file index



The FDUP$ request allows opening a file index which accesses the same file as a file index already OPENed by a program. The original file index specified in the packet is replaced by the newly assigned file index when the request is executed. Upon return from the FDUP$ request, either file index may be used to access the file. Note that since the file address pointer is associated with an open instance of a file and not the file itself, accesses performed with one file index will not affect the address pointer of the other. The ⟨access mode⟩ field is has the same meaning as in the OPEN$ request, and may be used to control whether reading and writing may be done through the duplicate

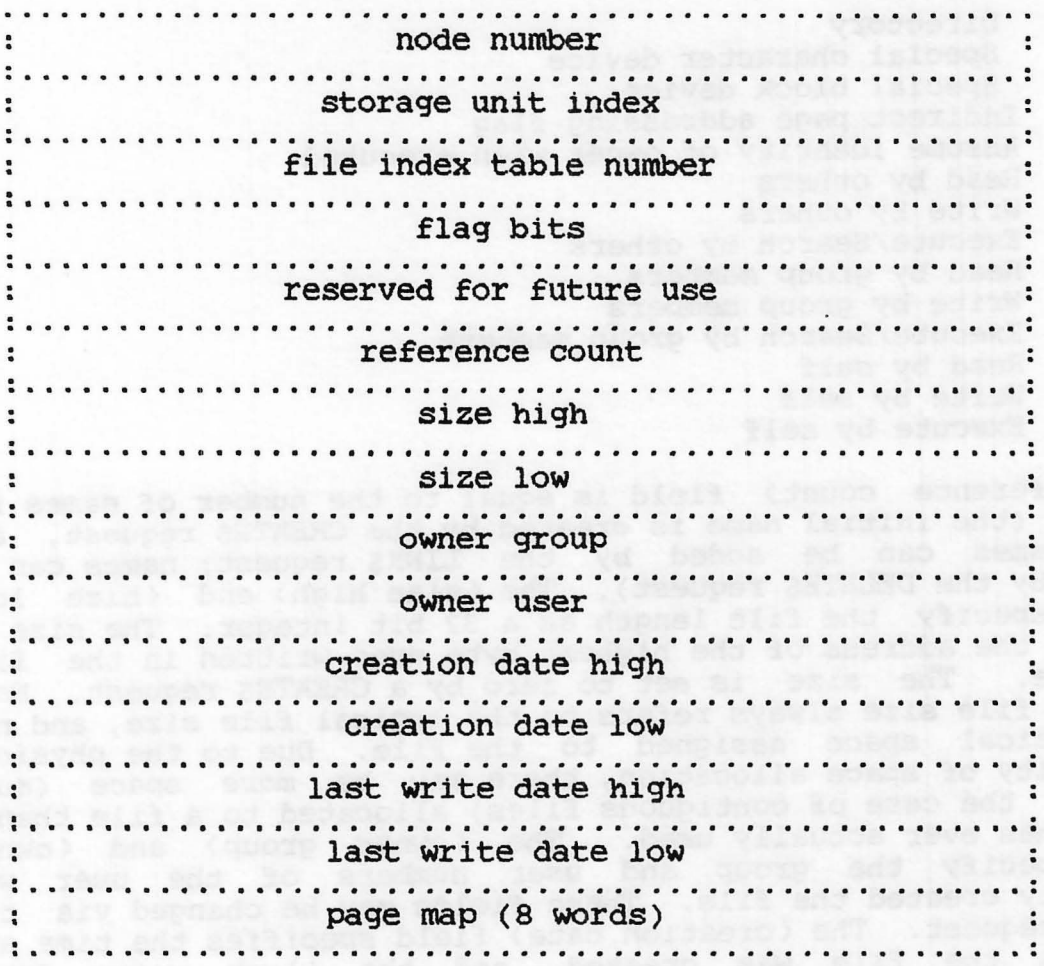
index regardless of the modes specified when the file was initially opened.

4.1.2.7. FSTAT$ (12) Obtain file status



The FSTAT$ request allows a program to read the file index table entry for a file. The ⟨name address⟩ field should contain the address of the file name, and the ⟨name length⟩ field should contain the length of the file name in bytes. The ⟨buffer address⟩ field must point to the first byte of the area in which the file index table entry will be stored, and the ⟨buffer length in bytes⟩ field specifies the length of that area. In the current release of the system, file index table entries are 44 bytes long, so if the entire entry is to be read, the buffer area should be that long. If the buffer length specified is less than the length of the entry, only the length requested will be transferred. In any case, the ⟨(bytes transferred)⟩ field will contain the number of bytes actually returned.

The format of the file index table entry returned by the FSTAT$ request is as follows:



The ⟨node number⟩ field specifies the number of the node within the resource sharing network to which the storage unit on which the file resides is connected. The ⟨storage unit index⟩ field is a code identifying the storage unit where the file resides. This binary value is equal to the explicit storage unit name (such as 1:) which can be used when referencing the file. The ⟨file index table number⟩ field will contain the unique code identifying the file on the volume on which it is stored. Multiple names for a file may be identified since they will always return the same ⟨node number⟩, ⟨storage unit index⟩, and ⟨file index table number⟩ values.

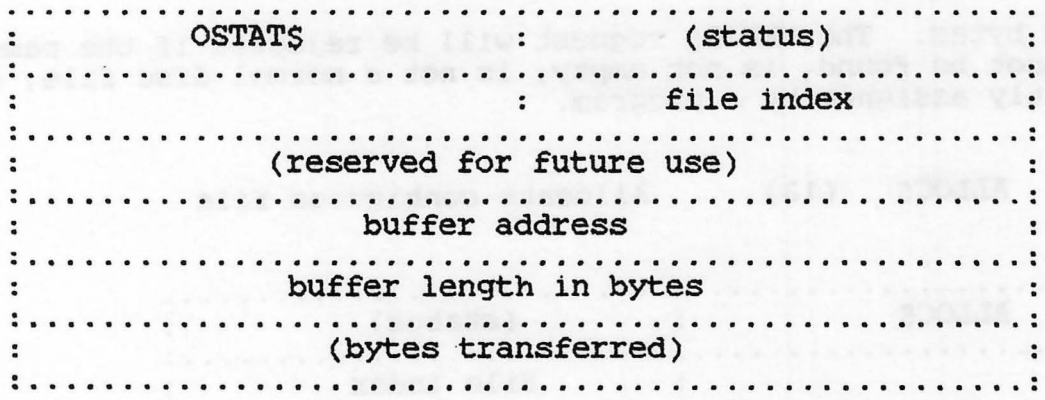
Flag bits are defined as follows (hex values):

```
8000      File is allocated
4000      Contiguous file
3000      File type
 0000      Normal disc file
```

1000	Directory
2000	Special character device
3000	Special block device
0800	Indirect page addressing flag
0200	Assume identity of owner when executed
0100	Read by others
0080	Write by others
0040	Execute/Search by others
0020	Read by group members
0010	Write by group members
0008	Execute/Search by group members
0004	Read by self
0002	Write by self
0001	Execute by self

The ‹reference count› field is equal to the number of names for the file (the initial name is created by the CREATE$ request, and other names can be added by the LINK$ request; names can be removed by the DELETE$ request). The ‹size high› and ‹size low› fields specify the file length as a 32 bit integer. The size is equal to the address of the highest byte ever written in the file plus one. The size is set to zero by a CREATE$ request. Note that the file size always refers to the logical file size, and not the physical space assigned to the file. Due to the physical granularity of space allocation, there may be more space (much more in the case of contiguous files) allocated to a file than a program has ever actually used. The ‹owner group› and ‹owner user› specify the group and user numbers of the user who originally created the file. These fields may be changed via the CHOWN$ request. The ‹creation date› field specifies the time and date when the file was created, and the ‹last write date› specifies the time and date when the file was last modified. These dates are stored as 32 bit integers in the same format as returned by the TIME$ request. The ‹page map› cells contain the information used to locate the physical storage assigned to the file. If the file is contiguous, word 0 will be the number of the first storage block assigned to the file and word 1 will be the number of blocks assigned to the file. If the file is not contiguous and the indirect page table bit in the flags word is zero, the page table words contain the block numbers for the file storage. If the indirect page table bit is nonzero, the page table contains addresses of blocks which contain pointers to the physical storage assigned to the file.

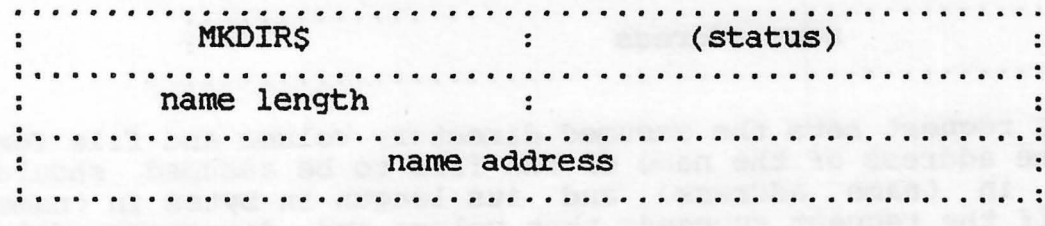
4.1.2.8. OSTAT$ (23) Obtain open file status



The OSTAT$ request allows a program to read the file index table entry for a file previously opened by the program. This request is exactly like the FSTAT$ request described above, and returns identical information in the buffer, except that it takes the ⟨file index⟩ of a file previously opened by the program rather than the name of a file which need not be previously opened. Refer to the description of the FSTAT$ request for information on the use of fields in the request packet and the meaning of information returned.

OSTAT$ is primarily used in subroutines which are passed the index of an open file, and need to determine information regarding the file. By using OSTAT$, they need not know the name of the file on which they are working. Also, if the user has previously opened the file, it is more efficient to use OSTAT$, since it does not have to look up the file name again as FSTAT$ would be forced to do.

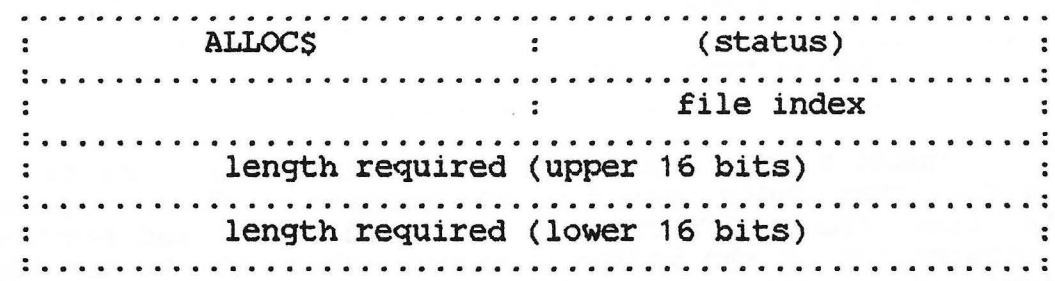
4.1.2.9. MKDIR$ (0A) Make file into directory



The MKDIR$ request converts an empty file (normally just created with the CREATE$ request) into an empty directory. The file is marked as a directory, and the self pointer (.) and parent directory pointer (..) files are created. The ⟨name address⟩ field specifies the address of the file name to be converted, and the ⟨name length⟩ field specifies the length of the file name

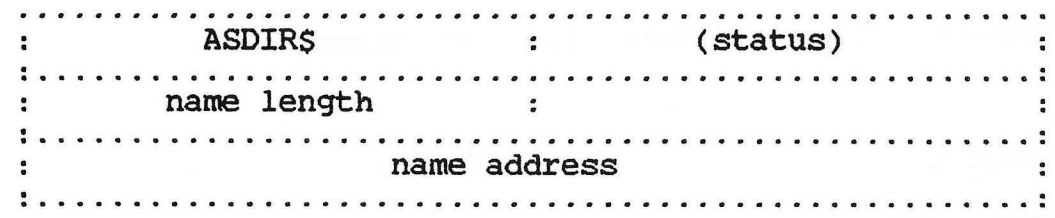
string in bytes. The MKDIR$ request will be rejected if the named file cannot be found, is not empty, is not a normal disc file, or is currently assigned by a program.

4.1.2.10. ALLOC$ (13) Allocate contiguous file



The ALLOC$ request attempts to allocate contiguous space for a file. Any space previously allocated to the file, whether contiguous or not, will be released. The ⟨file index⟩ must be a file currently open for writing, and the ⟨length required⟩ fields must contain the number of bytes to be allocated as a 32 bit integer. If there is insufficient contiguous space to satisfy the request, the request will be rejected with a bad status and no space will be allocated.

4.1.2.11. ASDIR$ (1A) Assume default directory



The ASDIR$ request sets the assumed directory volume and file for a user. The address of the name of the file to be assumed should be stored in ⟨name address⟩ and its length in bytes in ⟨name length⟩. If the request succeeds that volume and directory file will be assumed on all subsequent file operations taking a file name when no volume and/or leading slash in the file name is specified. The file name used with ASDIR$ must specify a file, not just a device or volume. If the root directory on a volume is to be assumed, the self name "." of that directory must be specified.

If ⟨name length⟩ is zero, the current assumed directory (if any) will be dropped. All files referenced by the user subsequent to such a call must contain an explicit volume and file name specification beginning with the root directory on the volume until another default directory is specified with ASSUME$.

4.1.2.12. CHACC$ (18) Change access/privacy modes

CHACC$	(status)
name length	
name address	
new privacy modes	

The CHACC$ request changes the privacy modes of a file. The address of the file name is stored in the ⟨name address⟩ field and the length of the name in bytes is stored in the ⟨name length⟩ field. The new mode bits are placed in the ⟨new privacy modes⟩ field. See the discussion of privacy mode bits in the description of the CREATE$ request for the meaning of the various bits. Only the owner of a file or a privileged user is permitted to change the privacy modes of a file.

4.1.2.13. CHOWN$ (19) Change file ownership

CHOWN$	(status)
name length	
name address	
new owner group	
new owner user	

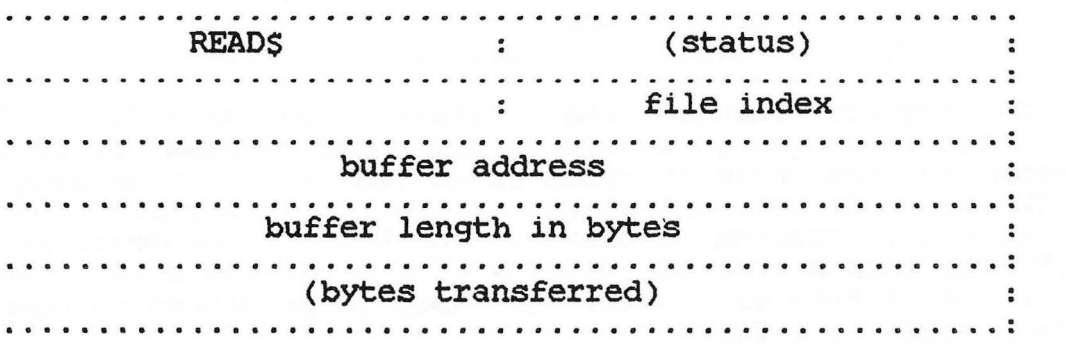
The CHOWN$ request, which may be made only by privileged users, allows the ownership of a file to be changed. The ⟨name address⟩ and ⟨name length⟩ specify the address and length of the file name to be changed. The ⟨new owner group⟩ and ⟨new owner user⟩ specify the group and user numbers for the file's new owner. Note that

since the ownership of a file is a property of a file and not of its (possibly multiple) names, changing the ownership of a file may result in users who have links to it in their directories not being able to assign (or delete) the file.

4.1.3. Input/Output

The Input/Output requests control transfer of data to and from files (including device files).

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



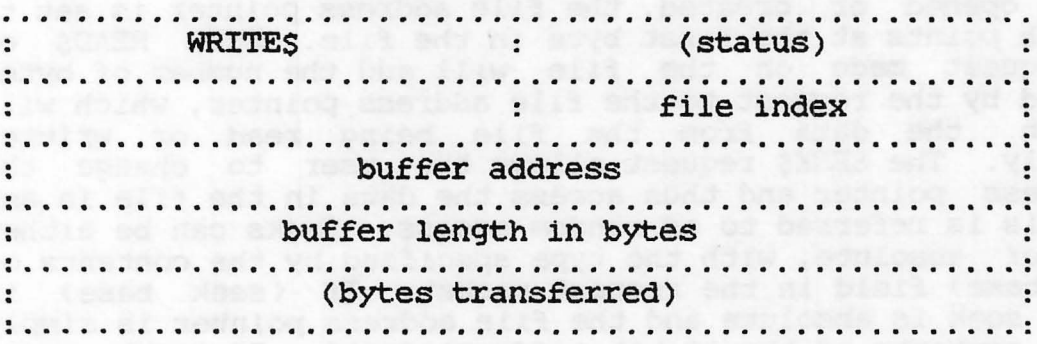
The next block of information from the file specified by ⟨file index⟩ is read into the buffer starting at the address specified by ⟨buffer address⟩. The length of the block read is specified by ⟨buffer length in bytes⟩. The actual length of the block read in will be stored in ⟨(bytes transferred)⟩. When reading from a disc file, the length read in will always equal the buffer length except when the read is truncated due to encountering either the end of the file or an I/O error. When reading from a device file, such as the user terminal, each READ$ request will transfer a logical record of information (one input line in the case of the user terminal). Input from the terminal is always terminated by a carriage return. The ⟨(status)⟩ field will be zero for normal completion, 1 if no data was transferred because end of file was encountered, and 2 if an unrecoverable I/O error occurred. Note that the end of file status occurs only when no data is transferred. A read that starts before the end of a file but is truncated at the end of file will receive a normal status. This condition can be tested for, if desired, by comparing the ⟨(bytes transferred)⟩ field with the ⟨buffer length in bytes⟩ field.

If a program is waiting for input from the user's terminal, and the Control C key is pressed, and the user has requested

notification of that event via TRAP$, the ⟨(status)⟩ field in the READ$ packet will be set to 22 and the ⟨(bytes transferred)⟩ will be set to zero. This enables a program returning from the TRAP$ to determine that no data have actually been obtained by the READ$.

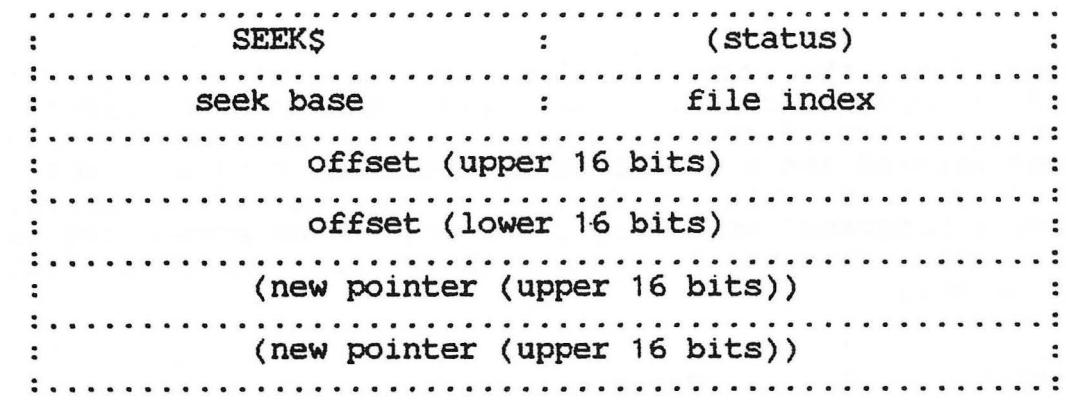
When reading from the user terminal, if the ⟨buffer length in bytes⟩ field is zero, no data will be read, but the ⟨(status)⟩ will be set to zero if data is available to be read and 1 if no data has been entered and a normal READ$ from the terminal would cause the program to wait for data to be entered. This special request permits programs performing other continuous processing to respond to input from the terminal without being forced to wait until it is entered.

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



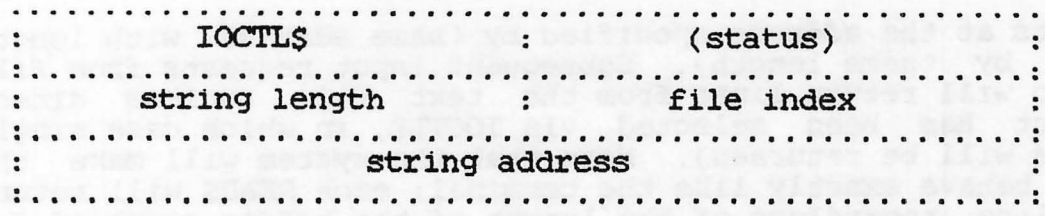
The information starting at ⟨buffer address⟩ with length ⟨buffer length in bytes⟩ is written to the file with the specified ⟨file index⟩. The number of bytes actually written to the file will be stored in the ⟨(bytes transferred)⟩ field. The ⟨(status)⟩ field will be set as for the READ$ request described above. Writing beyond the end of a normal file will cause automatic allocation of space to the file. Attempting to write beyond the end of a contiguous file will cause the write to be truncated. Note that since the end of file status is set only if no data were transferred, programs which write contiguous files should check the ⟨(bytes transferred)⟩ field against the ⟨buffer length in bytes⟩ field after each write and report an error if the two fields differ.

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



Files are normally processed sequentially. When a file is initially opened or created, the file address pointer is set to zero, which points at the first byte in the file. Each READ$ or WRITE$ request made on the file will add the number of bytes transferred by the request to the file address pointer, which will result in the data from the file being read or written sequentially. The SEEK$ request allows the user to change the file address pointer and thus access the data in the file in any order. This is referred to as random access. Seeks can be either relative or absolute, with the type specified by the contents of the ⟨seek base⟩ field in the request packet. If ⟨seek base⟩ is zero, the seek is absolute and the file address pointer is simply set to the contents of the 32 bit ⟨offset⟩ field. If ⟨seek base⟩ is 1, the seek is relative to the current position in the file, and the address pointer is set to the current value of the address pointer plus the ⟨offset⟩ field. If ⟨seek base⟩ is 2, the seek is relative to the end of the file, and the address pointer is set equal to the current file length plus the ⟨offset⟩ field. Note that the ⟨offset⟩ field may be either positive or negative (if negative, it should be the normal two's complement representation). At the completion of the SEEK$ request, the ⟨⟨new pointer⟩⟩ field in the packet will be filled with the resulting file address pointer after the SEEK$. Note that the pointer may be read by performing a seek with ⟨seek base⟩ equal to 1 and ⟨offset⟩ equal to zero.

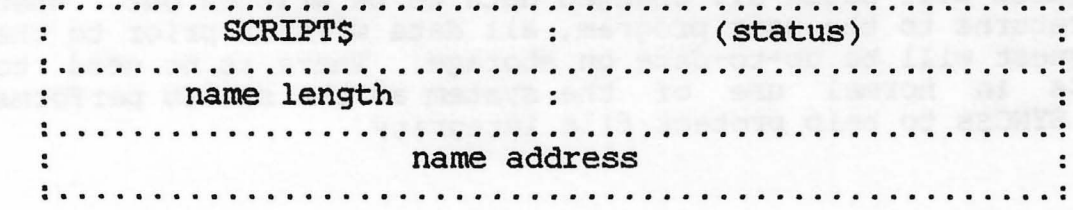
4.1.3.4. IOCTL$ (10) Set device file modes



The IOCTL$ request allows programs to communicate mode information to the device drivers which implement device files in the system. The request causes the device driver for the open file associated with ⟨file index⟩ to process the mode string whose start address is specified by ⟨string address⟩ with length equal to ⟨string length⟩. The action of the mode string depends on the device driver which processes it.

If sent to the user terminal device file (an open instance of CONS.DEV), only the first two bytes of the string are significant. The first byte is unused (reserved for a feature not yet implemented in the system). The second byte is the "raw mode" control byte. If zero, the user terminal will operate in normal buffered mode. In this mode the system assembles lines of text from the terminal, providing all the local editing features described earlier in this manual. A READ$ from the terminal will receive an entire line, and will not obtain the data until the user presses the RETURN key on the terminal. If the "raw mode" byte is set to 1, input from the terminal will be sent character-by-character to the active program. Each READ$ will return all characters typed since the last READ$ request. The system will not echo input to the display, permitting programs to echo whatever is desired. In raw mode, Control C is simply passed to the program as a data character, hence it cannot be used to interrupt a program that has turned on raw mode. Raw mode is automatically terminated when a program exits and the operating system prompt reappears.

4.1.3.5. SCRIPT$ (20) Set file for standard input



The SCRIPT$ request suspends the current source of standard input (read from file index zero) and attaches it to the text file whose

name starts at the address specified by (name address) with length specified by (name length). Subsequent input requests from file index zero will return lines from the text file (unless direct mode input has been selected via IOCTL$, in which case single characters will be returned). Note that the system will make the text file behave exactly like the terminal: each READ$ will return only one line, regardless of the length of the buffer supplied for READ$. In reading text files the system adheres to the convention that lines are delimited by CR (carriage return) characters and that the end of file is denoted by an EOT (end of transmission) character.

When the end of the file is reached, input will revert to the source in effect before the SCRIPT$ request was issued. Note that the user may "nest" SCRIPT$ requests, up to a limit defined when the system is generated. Since the system reads its commands from file index zero, SCRIPT$ may be used to cause the system to execute commands from a file as well as to cause programs to read from a file instead of the terminal.

4.1.3.6. SYNC$ (15) Write out all changed data

SYNC$	(status)

The Network Operating System automatically pages data from file volumes through buffers in memory. This paging greatly improves system throughput and enables the system to provide byte addressable files regardless of the physical block size of the devices on which the files are stored. This paging is totally transparent to user programs, but for some maintenance functions and special applications, it is desired to be able to force all data being held in memory out to the disc (for example, if two systems were communicating via a shared disc area, a system would have to know that the data had been physically written to the disc before it could tell the other system to read the data). The SYNC$ request will cause all changed data to be written out. When control returns to the user program, all data written prior to the SYNC$ request will be up-to-date on storage. There is no need to do SYNC$s in normal use of the system as the system performs periodic SYNC$s to help protect file integrity.

4.1.4. File and record locking

The FLOCK$ request, described below, coordinates access to files being shared by two or more programs running concurrently. The request allows both locking of all access to a file, and locking of individual records within the file (accomplished by a combination of FLOCK$ and file-dependent code in the user program).

4.1.4.1. FLOCK$ (24) Lock/Unlock access to a file

FLOCK$	(status)
function	file index

The FLOCK$ request controls access to a file. Files used with FLOCK$ must have been previously opened by the program. The ⟨file index⟩ field designates the file on which FLOCK$ is to operate. The action FLOCK$ takes depends on the contents of the ⟨function⟩ field. Each ⟨function⟩ is described below.

Function 1: Lock file, wait if busy
If the file is not currently locked by any user, the file is marked locked to the requestor and control is returned to the requestor. If the file is locked, the requestor's program is suspended and the lock request is queued. When the current locker of the file, and all requestors who were previously queued, have unlocked the file, the file will be marked locked to the requestor and the program will be reactivated.

Function 2: Unlock file
If the file has been previously locked by the requestor, it will be unlocked. All users waiting on a Function 4 request will be reactivated. If one or more users are waiting to lock the file with a Function 1 request, the first user in the queue will be given the lock on the file. If the user had not previously locked the file, a Function 2 request will result in an error status 11 in the ⟨(status)⟩ field, and no action will be taken.

Function 3: Lock, return status if busy
If the file is not locked at the time of the request, the lock is set. If the file is locked, the user is not queued. Instead, the ⟨(status)⟩ field in the packet is set to 19 (file or device busy) and control is returned immediately to the

program. This request may be used instead of the Function 1 request when the program wishes to take some other action if the lock is set rather than just waiting for it to be cleared.

Function 4: Unlock, wait for event
If the file is marked locked by this user, it is unlocked as described for the Function 2 request. After unlocking the file and activating users queued, the requestor is deactivated and queued on the file. The user will be reactivated the next time any user does an unlock (Function 2 or Function 4) on the file. This request is used in conjunction with the algorithms described below to implement locking of individual records in a file. If the user had not previously locked the file, the ⟨(status)⟩ field will be set to 11, and no action will be taken.

Function 5: Privileged clear lock
This request is identical to the Function 2 request, except that the requestor need not be the user who set the lock originally. A program must be privileged to perform this function; if executed by a nonprivileged program, status 11 will be returned in the ⟨(status)⟩ field and no action will be taken. This function is provided to allow recovery when a program locks a file and then aborts with the lock set, or when improper program logic causes a "circular hold" condition. It should never be necessary in normal operations.

The FLOCK$ request provides file-wide locking simply through the use of the Function 1 and Function 2 requests. A program wishing to lock a file should do a Function 1 request, do whatever it wishes to the file, then do a Function 2 request to unlock the file. Any number of programs running concurrently are assured that each has exclusive access to the file between its Function 1 and Function 2 requests.

For many applications, file-wide locking is enough. In some cases, however, especially in complex database systems, it is necessary to lock individual records or subsets of data within one file. When this is necessary, FLOCK$ Functions 1 and 4 may be used in conjunction with special algorithms to achieve the desired result. The user must design the database and decide where the locks are to be placed. The actual locks are simply data items in the file, which are read and written like any other data. To set a lock within the file the following algorithm should be used.

Algorithm L (Set a lock within a file). A lock within the file is set. If the lock is currently set, the program will wait until it is released, then set it and resume execution.
L1 (Set file-wide lock.) Perform FLOCK$ Function 1 on file.
L2 (Read lock from file.) Read record containing lock from

file.

L3 (Inspect lock.) Examine record lock item within record. If it is a 0, go to step L5.

L4 (Clear file-wide lock, wait for event.) Perform FLOCK$ Function 4 on file. After return from request, go to step L1.

L5 (Set record lock.) Set record lock item to 1.

L6 (Write record lock into file.) Write record containing lock back to file.

L7 (Clear file-wide lock.) Perform FLOCK$ function 2 on file.

After performing this algorithm, the lock on the record (or section, etc.) of the file will be set, and other users of this algorithm will be forced to wait until the lock is cleared by the execution of the following algorithm.

Algorithm U (Unlock record). The lock in a record is cleared.

U1 (Set file-wide lock.) Perform FLOCK$ Function 1 on file.

U2 (Read lock from file.) Read record containing lock from file.

U3 (Clear record lock.) Set record lock item to 0.

U4 (Write record lock into file.) Write record containing lock back to file.

U5 (Clear file-wide lock.) Perform FLOCK$ function 2 on file.

These algorithms assume that all the record lock items in the file have been initially set to zero. It should be clear from an examination of the record lock and unlock algorithms that the file-wide lock is set only momentarily when checking or changing the record lock. Thus, two programs which lock different records will not have to wait for one another, but programs wishing to access the same record will be properly queued.

Since the record locks are simply data in the file, if a program aborts while a record lock is set, other programs using Algorithm L will wait forever, since the record lock will never be cleared. Consequently, when implementing an on-line database system, it is useful to have a privileged utility which can be used to clear record locks. Obviously, this program should be used with great care, ideally when all on-line work has been shut down.

Users planning to use record locks should be extremely careful about locking more than one record at a time. If multiple records are locked, and great care is not taken in the design of the application, it is possible to get into a "circular hold" condition, where neither program can continue. If, for example, there are two locks in a file, Lock 1 and Lock 2, and two programs try to set them in opposite order (that is, Program 1 sets Lock 1,

then Lock 2, and Program 2 sets Lock 2, then Lock 1) it is possible to get into a circular hold where Program 1 has set Lock 1, Program 2 has set Lock 2, and both are waiting for each other's lock to be cleared. Since both programs will only clear the locks after setting both, there is no exit from this condition. The literature on detecting and avoiding this problem is extensive, and this is not the place to discuss it, but a word to the wise will hopefully be sufficient: don't set multiple locks at once unless you know what you are doing.

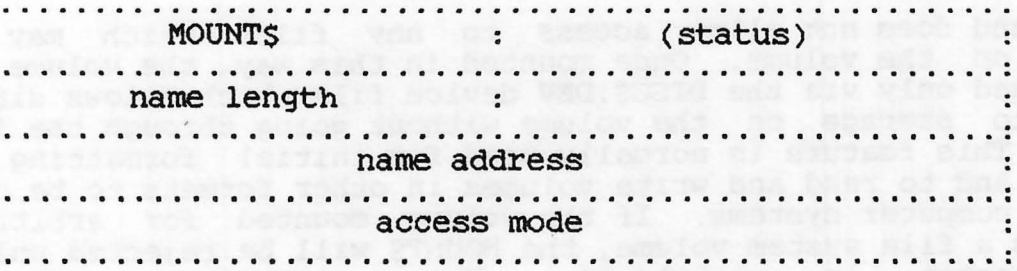
The locking done by FLOCK$ affects only other FLOCK$ requests; it does not prevent reads and writes from being done on the file. Hence, if FLOCK$ is being used to protect access to a file, all programs using the file must use FLOCK$. An FLOCK$ lock may remain set across a close and subsequent reopen of a file. If no other users are waiting on the lock, the Function 2 request to unlock the file will be rejected if the file has been closed and reopened, but the program may, in this case, ignore the rejection as the desired action (clearing the lock) has already been taken anyway.

Setting and clearing file-wide locks with FLOCK$ is an extremely fast operation involving no I/O to the file or directory. Hence, programs should freely set and clear the lock when required rather than trying to minimise the number of locks and unlocks at the cost of complicating the program. FLOCK$ works only on normal files stored on a storage unit; it may not be used on device files. Since memory files are a true storage unit, FLOCK$ may be used on them without restriction.

4.1.5. Volume maintenance requests

These requests control the status of storage volumes and the physical devices on which they are mounted.

4.1.5.1. MOUNT$ (16) Mount volume on storage unit



The MOUNT$ request causes a volume to be logically mounted, that is, made available for use through the system. The ⟨name address⟩ field points to a string specifying the device and volume name (which is described below), and ⟨name length⟩ specifies the length of that string. The ⟨access mode⟩ field specifies restrictions on the type of accesses which may be made to the volume during this mount. The ⟨access mode⟩ field is the sum of the desired modes from the following table:

4	Reading is to be permitted
2	Writing is to be permitted
1	Execution of programs is to be permitted

The string used with MOUNT$ has one of the three following formats:

⟨device⟩:⟨volume⟩

This format requests the mounting of a specific volume on a device. ⟨device⟩ is the device name (normally 1, 2, 3, etc.), and ⟨volume⟩ is the name of the volume to be mounted. If this form of the request is used, and the volume named in the request does not agree with the name of the volume mounted on the device, the request will be rejected with a status of 20. For example, to mount the volume "PAYROLL" on disc drive number 3, the parameter string would be "3:PAYROLL".

⟨device⟩:

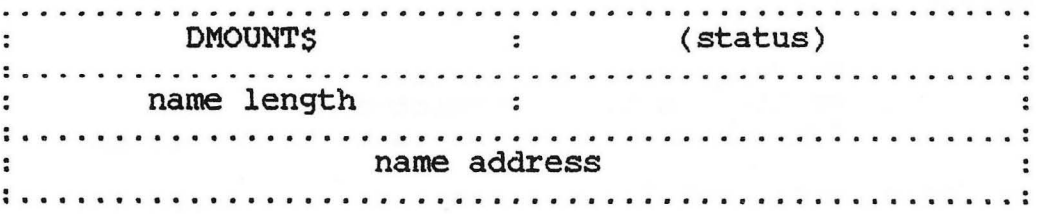
This form of the MOUNT$ request will mount whatever volume is physically present on the device, but does require that the volume be a properly-formatted file system volume. This request may be used whenever volume verification is not required. Files on the volume may be accessed by volume name after mounting, even though the volume name was not given in the MOUNT$ parameter string.

⟨device⟩:*

This form of MOUNT$ mounts a volume for "arbitrary format access". It does not require that the volume be formatted for file system

access, and does not allow access to any files which may be present on the volume. Once mounted in this way, the volume may be accessed only via the DISC$.DEV device file which allows direct access to storage on the volume without going through the file system. This feature is normally used for initial formatting of volumes, and to read and write volumes in other formats to be used on other computer systems. If the volume mounted for arbitrary access is a file system volume, the MOUNT$ will be rejected unless the requestor is privileged. This restriction prevents unauthorised destruction of system files through arbitrary access to them.

4.1.5.2. DMOUNT$ (17) Dismount storage volume

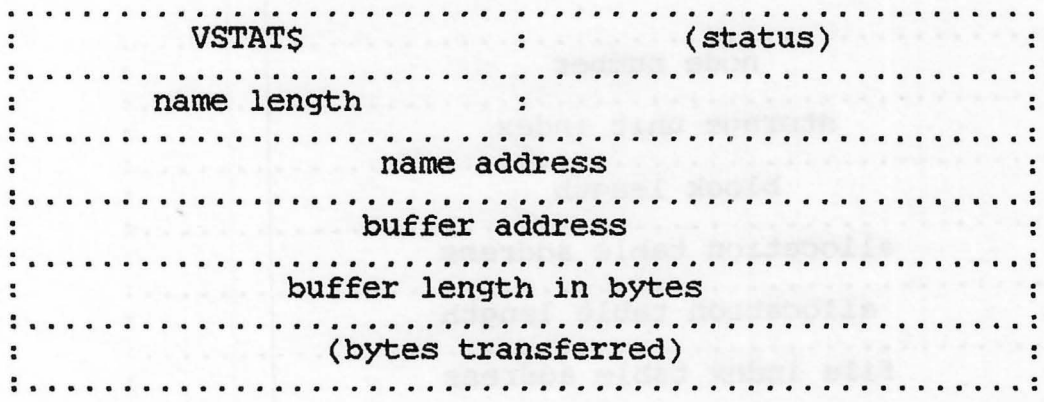


The DMOUNT$ request dismounts a volume from the storage unit on which it has been mounted. The ⟨name address⟩ specifies the address of a string of length ⟨name length⟩ which is of the form:

⟨device/volume⟩:

Note that either the device name or volume name may be specified, and that the name must be followed by a colon. A volume may be dismounted only when no files on it are currently open. If a DMOUNT$ is attempted while files are open on the volume, the request will be rejected with a status of 19.

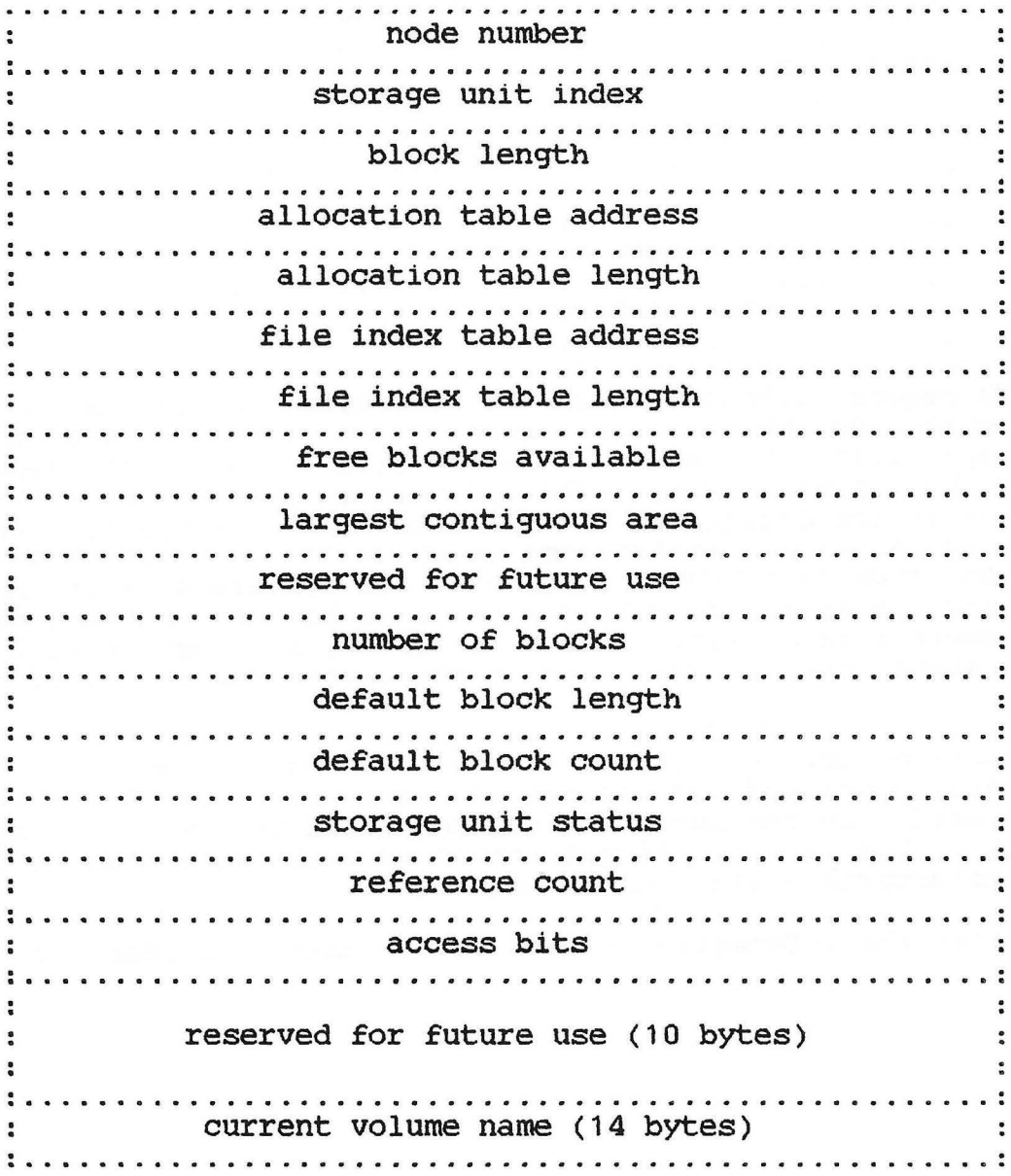
4.1.5.3. VSTAT$ (1F) Obtain unit/volume status



The VSTAT$ request allows the user to determine the status of a storage unit and the volume mounted on it (if any). The ⟨name address⟩ specifies the address of a file name, with length specified by ⟨name length⟩. The information returned is for the unit and/or volume designated by that file name. If the file name contains an explicit device name (such as "1:"), or an explicit volume name (such as "SYSVOL:"), it need not contain a file name portion, and the device or volume named will be used. If the name does not contain an explicit device or volume, information will be returned about the volume on which the user's default directory resides.

The information returned will be stored in a buffer starting at the address specified by ⟨buffer address⟩. No more information will be stored than the buffer length specified by ⟨buffer length in bytes⟩. The actual length returned will be stored in the ⟨(bytes transferred)⟩ field of the packet.

The format of the information stored in the user's buffer is as follows:



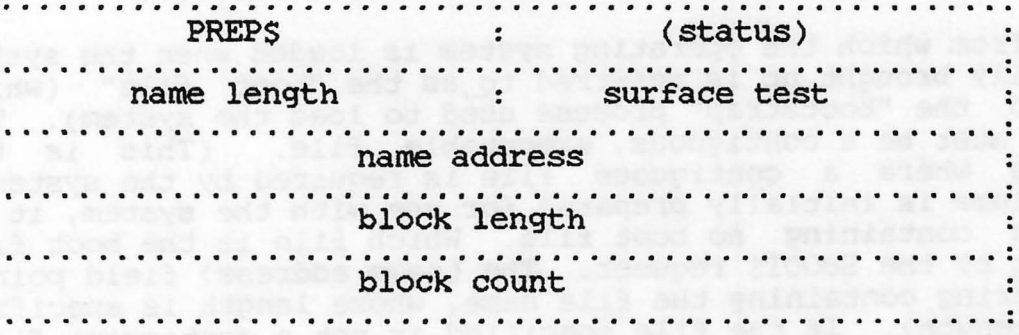
The ⟨node number⟩ field specifies the node number in the resource sharing network which supports the storage unit selected. The ⟨storage unit index⟩ field is the integer code for the storage unit (1, 2, etc.). The ⟨storage unit status⟩ field will be zero if the unit is idle (no volume mounted), one if a normal file system volume is mounted, and two if an arbitrary format volume is mounted (asterisk specified as volume name).

The ⟨block length⟩ field specifies the length of allocation blocks on the device, in bytes, and ⟨number of blocks⟩ specifies the

number of such blocks provided on the device (or volume). The ⟨default block length⟩ and ⟨default block count⟩ fields specify the size in bytes and number, respectively, of blocks assumed to be present when a volume is prepared for access on this storage unit (PREP$) and no explicit specifications are given for these values in the PREP$ request. ⟨reference count⟩ is the number of files currently open on the storage unit, and ⟨access bits⟩ are the modes of access permitted to the unit, as specified in the MOUNT$ request packet.

The following fields will be returned only if the storage unit currently has a valid file system format volume mounted (⟨storage unit status⟩ field equal to one). The ⟨allocation table address⟩ field specifies the start block number of the allocation bit table, and ⟨allocation table length⟩ specifies the length of that table in terms of blocks of length specified by ⟨block length⟩. ⟨file index table address⟩ specifies the start block number of the file index table, and ⟨file index table length⟩ specifies the length of that table in blocks. The ⟨current volume name⟩ field is the 14 character name of the volume currently mounted on the storage unit. The number of currently unallocated blocks available for assignment to files created on the volume is given in the field ⟨free blocks available⟩, and the largest contiguous unallocated area is specified, in blocks, by the field ⟨largest contiguous area⟩.

4.1.5.4. PREP$ (21) Write directory on volume



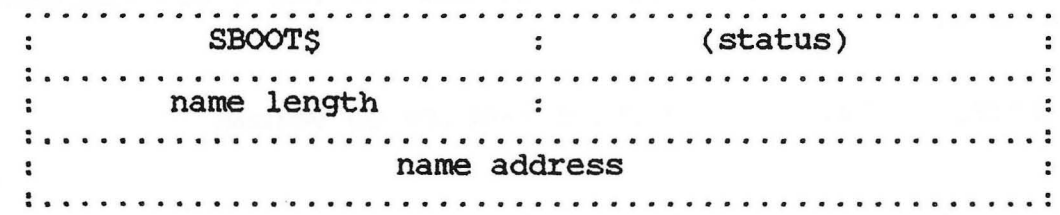
The PREP$ request creates an empty file system on a volume, optionally performing a complete surface test and removing any bad blocks from eligibility for allocation. PREP$ may be performed only by a program executing under group number 0 and user number 666. Any other user attempting to perform a PREP$ will receive a protection reject status. The storage unit containing the volume to be PREPed is specified by the string whose address is given by ⟨name address⟩ and length by ⟨name length⟩. The string must

contain an explicit storage unit name (such as "1:"). That unit must contain a volume mounted for arbitrary access (volume name of "*" in the MOUNT operation).

The ⟨block length⟩ field in the packet specifies the block size in bytes to be used in allocating space on the volume, and ⟨block count⟩ specifies the number of such blocks present on the unit. If one or both of these fields are zero, the default values configured for the storage unit will be used. Note that the ⟨block length⟩ and ⟨block count⟩ specified must be compatible with the hardware driver for the storage unit.

If the ⟨surface test⟩ field contains 1, all blocks on the unit will be written, and any which cannot be read back successfully will be removed from the list of blocks available for allocation to user files. If ⟨surface test⟩ is 0, this will not be done, and all storage blocks will be assumed to be good. The complete surface test may take a large amount of time to complete.

4.1.5.5. SBOOT$ (14) Set boot file



The file from which the operating system is loaded when the system is initially brought up is referred to as the "boot file" (which refers to the "bootstrap" process used to load the system). The boot file must be a contiguous, executable file. (This is the only case where a contiguous file is required by the system.) When a volume is initially prepared for use with the system, it is marked as containing no boot file. Which file is the boot file may be set by the SBOOT$ request. The ⟨name address⟩ field points to the string containing the file name, whose length is specified by ⟨name length⟩. If the file specified is not a contiguous file, the request will rejected with a status of 14. The SBOOT$ request may be made only by privileged users.

4.1.6. System environment requests

The environment requests allow specification and retrieval of information regarding the environment in which the system exists.

While these requests are always provided in the system, the validity of the information they return depends upon the presence of certain hardware and support software which may not be provided in all configurations which support the Network Operating System.

4.1.6.1. TIME$ (1D) Return current time

TIME$	(status)
(time (upper 16 bits))	
(time (lower 16 bits))	

The ⟨(time)⟩ fields in the packet will be filled with the current date and time value. The system represents date and time as a 32 bit unsigned integer representing the number of seconds elapsed since 00:00 GMT January 1, 1950. For compatibility in distributed networks which span time zones, system time is normally maintained in Greenwich Mean Time. System maintenance personnel may change this standard, so you should check with with system management first. The format used for time allows resolution of times to the second for all dates from 1950 to 2086.

4.1.6.2. STIME$ (1E) Set system time

STIME$	(status)
time (upper 16 bits)	
time (lower 16 bits)	

If the caller is privileged, the 32 bit ⟨time⟩ will be used to set the system time and date clock. The value stored in the ⟨time⟩ field should be in the same format as the system returns for the TIME$ request (see above). STIME$ requests from nonprivileged callers will be rejected.

4.1.7. 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
9	Memory INODE table full. (The combination of user and system files open is too large.)
10	File path name specifies a non-directory as a name other than the last in the path.
11	This request was rejected because completing it would violate system security.
12	Storage exhausted on unit
13	Maximum files already allocated on volume
14	Improper file type for requested function
15	Operation would result in two names for directory file
16	File must be void for this function
17	Attempt to create link from one volume to another
18	File already exists
19	File / device busy
20	Wrong volume mounted on device
21	Invalid volume / not file system format
22	SCRIPT$ files too deeply nested
23	Control C aborted JSYS
24	Too many processes active to create new one

4.2. Program execution environment

When a program is given control by the Network Operating System, 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.

4.2.1. Memory allocation

The standard starting address of programs run under the system 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 Network Operating System and must not be modified by programs. The area of memory from the end of the user program to the end of the user space is available for use by the program (for example, as a buffer pool). The starting address and length of this area can be determined by use of the MEM$ system call.

4.2.2. Initial workspace

When a program is given control after being loaded by the system, 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.

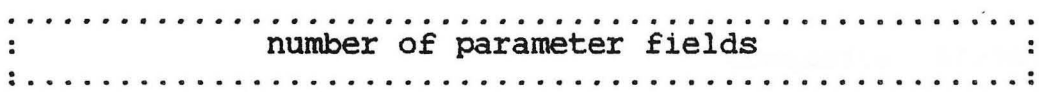
4.2.3. Program parameter string

The command line used to invoke a program may be read by a program by using the pseudo device file "PARAM.DEV". This file may be opened like any other file. When a READ$ request is issued to the file, it will return the text of the command line used to call the current program, terminated by a carriage return. The ‹(bytes transferred)› field in the READ$ packet will be set to the length of the command (including the terminating carriage return). Note that the Network Operating System returns the entire command line, while the Disc Executive returns only the portion of the command

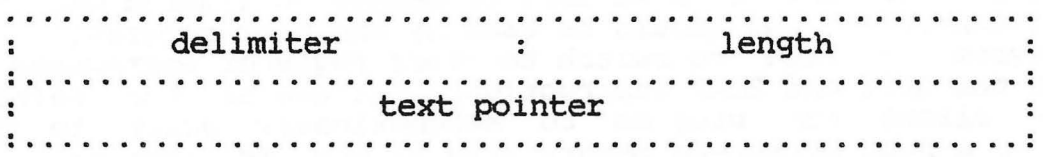
following the program name. This change has been made as a result of user requests to be able to retrieve the name of the current program. Existing programs may be easily converted to run under either system by simply ignoring characters in the string returned from PARAM.DEV until a space is found (under the Disc Executive, the PARAM.DEV string will always start with a space). The parameter string may be read any number of times. The data returned will always be identical.

4.2.4. Program parameter table

When the system loads a program, it will scan the command line used to invoke the program and build a parameter table which is stored following the program in memory. When the program receives control, register R1 in the initial workspace will contain the address of this parameter table. The first word of the parameter table is the number of parameter fields found on the command line (including the name of the program as the first field). Hence the table begins:



Immediately following this word are one or more parameter pointer entries. Each parameter pointer entry has the format:



where ⟨delimiter⟩ is the ASCII code for the delimiter that ended this parameter field, ⟨length⟩ is the length of the text of the field in characters, and ⟨text pointer⟩ is the address of the text for the field. The text of the field stored starting at the address in ⟨text pointer⟩ will be translated to upper case, will start on a word boundary, and will always be followed by a NUL (zero code) for those who do not like to count characters. The following characters serve to delimit fields:

, = ( ) space

Leading spaces before the next field will be ignored. If two field delimiter characters occur in a row, a zero length field will occur. The delimiter character for the last field will be zero.

To illustrate the format of the program parameter table, the following is the assembly language code that would generate the same parameter string as the system would for the command:

```
FARBLE widget.cob=ZONK.uLc(2,19),print.dev

        data    7                       number of fields
.
        byte    " ",6                   field 1
        data    f1
.
        byte    "=",10                  field 2
        data    f2
.
        byte    "(",8                   field 3
        data    f3
.
        byte    ",",1                   field 4
        data    f4
.
        byte    ")",2                   field 5
        data    f5
.
        byte    ",",0                   field 6
        data    0
.
        byte    0,9                     field 7
        data    f7
.
f1      text    "FARBLE"
        byte    0
        even
f2      text    "WIDGET.COB"
        byte    0
        even
f3      text    "ZONK.ULC"
        byte    0
        even
f4      text    "2"
        byte    0
        even
f5      text    "19"
        byte    0
        even
f7      text    "PRINT.DEV"
        byte    0
        even
```

The program parameter table will be stored immediately following the end of the program, and will be included in the program length

by the system. The MEM$ request will return the first free byte following the parameter table. Programs are free to use the parameter table area for scratch space. The address passed in R1 may be used as the first free byte address in that case.

4.3. System subroutines

The Network Operating System makes a set of generally useful subroutines available to programs running under its control. These subroutines are used within the system 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    "1:SOURCE/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.

4.3.1. 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.

4.3.2. 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.

4.3.2.1. 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.

4.3.2.1.1. 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.

4.3.2.1.2. 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.

4.3.2.1.3. 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.

4.3.2.2. 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.

4.3.2.2.1. 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.

4.3.2.2.2. 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.

4.3.2.2.3. 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.

4.3.2.3. 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.

4.3.2.3.1. 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.

4.3.2.3.2. 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.

4.3.2.3.3. 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$.

4.3.2.4. 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.

4.3.2.4.1. 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.

4.3.2.4.2. 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.

4.3.2.5. 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.

4.3.2.5.1. 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.

4.3.2.5.2. 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$, then overlaying the number in the field with EDECF$.

4.3.2.5.3. 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.

4.3.2.5.4. 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.

4.3.2.6. 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 &.&'
```

4.3.3. 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     "1:SOURCE/PRIMW$"
```

4.3.3.1. 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.

4.3.3.1.1. 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.

4.3.3.1.2. 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.

4.3.3.1.3. 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.

4.3.3.1.4. 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.

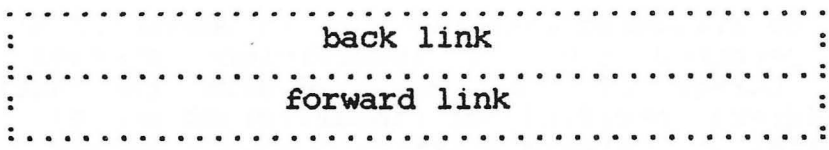
4.3.3.1.5. Buffer allocation errors

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$.
012	Backpointer in next buffer was bad. This will result if the program using the buffer stored off the end of the buffer, and may also result if a bad address if passed to BREL$.

4.3.3.2. 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.

4.3.3.2.1. 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.

4.3.3.2.2. 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.

4.3.3.2.3. 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$.

4.3.3.2.4. 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
.       .
.       .
.       .
```

5. Optional system features

Some of the capabilities of the Network Operating System depend on the presence of certain hardware or software components whose expense or complexity are not desirable in all configurations of the system. This section describes those features of the system which may be configured in a system or not be provided at the discretion of the person who generates the system. Obviously, it would be wise to discuss the configuration with that person before undertaking to use any of the features described herein. Note also that programs which use these features will not run on all configurations of the Network Operating System and thus are unwise to include in programs which are intended for wide distribution.

5.1. Printer driver

The system allows configuration of any number of printers. These printers may be accessed either directly through device files (discussed in this section) or by queueing files to an offline printer driver (discussed in a later section). Printer device files used for direct access to the printer are usually named:

PRINT.DEV	For the first one
PRINT2.DEV	For the second one
PRINT3.DEV	...et cetera

The names given to printers are defined when the system is generated, so consult the person responsible for system generation at your installation for details of the configuration you are using.

5.1.1. Opening the printer

Before a printer may be used by a program, it must be opened via the OPEN$ request. This request will return a busy status if the printer is in use by another user or by the offline print driver. Otherwise, the printer will be assigned to the user.

5.1.2. Output to the printer

Output is sent to a printer with the WRITE$ request, specifying the file index returned by the OPEN$ for the printer device file. There is no restriction on the length of the output buffer, which may contain as many lines of output (delimited by carriage returns) for the printer as desired. Line feeds may be used to skip blank lines, and a form feed character will cause the paper to be ejected to the next page. These characters will work even if the printer does not recognise them, since the driver will expand these characters to the sequences required by the printer to accomplish the desired function, if necessary.

5.1.3. Closing the printer

When a printer file is closed via CLOSE$, the printer will skip to the top of the next page and the printer will be released from the requesting program and made available to others. If a program neglects to close the printer before terminating, the system will

automatically release the printer when the program finally exits.

5.2. Offline file printing

In addition to direct access to printers through their device file names (discussed in the preceding section), the system also allows printers to be used to print files from storage volumes in an offline or background mode. Whether this feature is provided depends on the system configuration, and all printers need not be provided with this method of access. This facility is frequently referred to as a "print spooler", in which the word "spool" is an acronym for Simultaneous Peripheral Operation On Line.

Offline printer access is handled through the system's device file mechanism. Just as there is a device file associated with the printer for direct access, there may be a device file configured for offline access. While this device file may have any name at all, Marinchip Systems recommends that the name be the same name given to the corresponding printer device file with the ".DEV" replaced by ".OFF". Hence the offline access file names in a system might be:

```
PRINT.OFF       For the first printer
PRINT2.OFF      For the second one
PRINT3.OFF      ...and so on
```

These device files are used in a very different manner than the device files associated directly with the printer. While one sends DATA to a printer directly, one sends FILE NAMES to its offline access file. These file names are queued and the data in the files are printed on the printer.

5.2.1. Opening the offline access file

The offline access file for a printer is opened like any other device file. The file index returned in the OPEN$ packet is used to later send information to the file.

5.2.2. Output to the offline access file

Output to the offline access file consists of lines, where each line names a file to be printed on the printer associated with the offline access file. Each line is terminated by a carriage return, and the format of information in each line is:

```
   ))file name
or ##file name
```

Lines which do not begin with either the two character sentinel ")>" or "##" will be ignored. The file name following the sentinel may be any non-device file name. The file name is looked up in the context of the user's working directory at the time the line is written to the offline printer file. If the file name is preceded by the sentinel ")>", the file will be printed on the printer and when completed no special action will be taken. If the sentinel "##" is used, the file will be deleted after completion of printing. If the permissions on the file named would deny the user permission to read the file, it will not be printed, and if those permissions would not allow the user to delete the file himself, it will not be deleted after printing.

Any number of lines requesting file printing may be sent with one WRITE$ to the offline print file. There is a limit on the number of files which may be queued to a printer. If this limit is exceeded, a busy status will be returned for all writes to the offline printer file until the printer completes a file and reduces the queue length by so doing.

5.2.3. Closing the offline printer file

After sending file names to the offline printer file, the user may close it via CLOSE$. If this is not done, the system will automatically close the file when the user program terminates.

5.2.4. Offline and direct print contention

Files may be queued to a printer via the offline file at any time (except if the queue length is at its maximum limit). If the printer is in use by a user through the direct access device file, the offline print driver will wait until the user relinquishes control. The offline driver will then assign the printer to itself. While it is using the printer, users who try to access the direct device file will receive a busy status. Between files, the offline driver will release the printer for a brief interval to give user programs a chance to assign the printer directly. When all files have been printed, the offline driver will release the printer until more work is queued for it.

5.3. Background batch capability

Normally there is a one to one relationship between terminals used to access the system and user spaces in which programs are run from those terminals. If the background batch feature is configured, additional user spaces may be configured which are not associated with any terminal. These user spaces can be allocated for the use of programs running in other user spaces through the device file mechanism, and used to run other work concurrent with work done from the terminal.

Background batch is normally used in two ways. First, it can be used to start up a lengthy stream of work which can be done without interaction while the user continues to use his terminal normally. Second, it may be used by a program to call other programs as subroutines, interacting with them exactly as a user would from a terminal.

5.3.1. Batch space nomenclature

The background batch feature is implemented through the system's device file mechanism, and consequently, there is a device file name associated with each background batch space. The names given are up to the person who generates the system, but the names recommended by Marinchip Systems are:

BATCH.DEV	For the first one
BATCH2.DEV	For the second one
BATCH3.DEV	...and so on

5.3.2. Opening a batch space

When an OPEN$ is executed for a background batch space, the system will first check if that space is already in use. If so, the open will be rejected with a busy status (even if the space was opened by the program now trying to "re-open" it). If the space is idle, it will be marked assigned to the requesting program, and the group and user ID will be set to those of the requestor. The batch space will also be given the same working directory as the user who opens it, so that commands executed in that space will be interpreted the same as commands performed by the requesting user from the terminal.

5.3.3. Output to a batch space

When a WRITE$ request is used to write data to a batch space, that data is queued as input to the space, just like input from a terminal is queued to a normal interactive user space. Since all batch user spaces start in system command mode, the first line written to a space is normally a command to be executed. The system allows the program operating the space to get ahead of the program executing in the space, but if more than a limited number of WRITE$ requests are made without having the data read by the batch space, the writing program will be deactivated until the batch space catches up.

A program interrupt identical to typing a Control C on a terminal may be sent to a batch space by writing a single character with decimal value 3 (which is ETX, or Control C) to the batch space with WRITE$. Note that like a Control C from a terminal, this causes all queued input and output to be discarded.

5.3.4. Input from a batch space

When the file index of a batch space file is used with READ$, output from the program in the batch space is returned to the program controlling the batch space. Each read request returns one line of output from the program in the batch space. The batch space is allowed to get ahead of the controlling program, but after a limit is reached, it will be deactivated until the queued lines are read by the controlling program.

If a READ$ is done on a batch space and no output is available, the program doing the READ$ will wait until output becomes available, except if the program in the batch space is waiting for input. If so, the READ$ will receive an EOF status (1), which informs the controlling program that output must be sent to the batch space before any more results will be created to read.

5.3.5. Closing a batch space

When a CLOSE$ is done on a batch space device file, any program currently active in the space will be terminated, and any files open will be closed. The assumed directory will be dropped, and the space will be deassigned from the user who opened it and made available to other users.

6. System utility programs

The Network Operating System 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 user terminal. 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.

These programs are supplied as files in the directory:

BIN

on the system volume. Additional utility programs may be added by the user simply by adding them to this directory.

6.1. ACCESS - Set file privacy modes

The ACCESS command is used to specify the access permissions for a file. ACCESS may also be used to cause the execution of a program to assume the group and user identity of the owner of the file containing the program. The format of the command is:

```
ACCESS ⟨file⟩([*]⟨modes⟩),...
```

where ⟨file⟩ is the name of the file to be changed. If an asterisk appears inside the parentheses, then the assume group and user mode will be set. The ⟨modes⟩ are specified as:

```
⟨global⟩-⟨group⟩-⟨owner⟩
```

where ⟨global⟩ are the permissions for users with a different group than the owner of the file, ⟨group⟩ are permissions for users other than the file owner with the same group number, and ⟨owner⟩ are the permissions for the owner of the file. The permissions are formed from zero or more of the following letters:

R	Read permitted
S	Search (directory lookup) permitted
W	Write permitted
X	Execution permitted

The X and S modes are identical: the same mode means search permitted for a directory file and execution permitted for a normal data file.

For example, to change the file TESTVER/COBOL so that all users can execute it, members of the owner's group can read or execute it, and the owner can read, write, and execute it, one would use:

```
ACCESS TESTVER/COBOL(X-RX-RWX)
```

Only the owner of a file or a privileged user may change the file access modes using ACCESS.

6.2. ALIAS - Create alternate name for file

The ALIAS command creates an alias for a file, that is, a new name which refers to the same file as an existing file name. The command used is:

ALIAS ⟨new name⟩=⟨old name⟩,...

where ⟨new name⟩ is the new name to be created and ⟨old name⟩ is the name of the existing file. Both ⟨old name⟩ and ⟨new name⟩ must refer to the same volume, and ⟨new name⟩ may not already exist. The ALIAS command will not affect ⟨old name⟩.

6.3. 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.

6.3.1. 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 user terminal. If the assembly listing is sent to a disc or device file other than the terminal, lines with errors will still be logged on the terminal.

6.3.2. For more information

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

6.4. ASSUME - Set assumed volume and directory file

The ASSUME command allows specification of the default volume and directory for user file references. The command is:

```
   ASSUME ⟨file⟩
or ASSUME ⟨volume/device⟩:
or ASSUME
```

where ⟨file⟩ is the name of the directory to be assumed (the specification may contain an explicit volume or device name, as in any file name). The named ⟨file⟩ must be a directory. Following the ASSUME command, any file name used by the user which does not contain an explicit volume or device name will be assumed to refer to the volume of the assumed directory file, and any file name referenced on that volume which does not start with a leading slash (/) will be assumed to reside in the assumed directory file. If simply a volume or device name is specified, the root directory on that volume or device will be assumed. If no specification is given, the current assumed directory will be dropped and no new directory will be assumed. All file references made subsequently must be completely explicit including the volume name and the complete file name starting at the root directory on the volume, until another directory is ASSUMEd.

6.5. 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 simply by typing the file name. Consult the person responsible for software maintenance at your installation to determine which BASIC is available on the machine you use.

6.5.1. 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.

6.5.2. For more information

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

6.6. 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.

6.6.1. 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
```

6.6.2. Copying contiguous files

If the ⟨output⟩ file specified already exists, and is a contiguous file, BCOPY will not re-create the file as it would if the file were not contiguous. This allows one to BCOPY into a contiguous file without causing the contiguous space to be released, and hence the file made non-contiguous. If the file being copied into the contiguous output file is larger than the contiguous space allocated, an error message will result. The file must be re-allocated larger before BCOPY will be able to copy the data into it.

6.6.3. 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.
Either the input file name could not be found in the directory, or the output file name could not be created.

6.7. BOOTFILE - Set system load file

The BOOTFILE command allow selection of the file from which the Network Operating System is loaded when the system is initialised.

BOOTFILE ‹file›

The BOOTFILE command will mark the designated ‹file› as the boot file on the volume on which it resides. If that volume is then placed in the system load device and the system boot procedure followed, the system will be loaded from that file. The ‹file› named must be a contiguous file, as system loading is only possible from contiguously stored files.

The BOOTFILE command may be used only by privileged users.

6.8. 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.

BRAINSTORM is supplied by Marinchip Systems with the Network Operating System, and is normally available for use by all users. Since it is extremely costly to run in terms of computer time, and hence degrades performance in multi-user configurations, many installations restrict the use of BRAINSTORM to special personnel. Contact those responsible for system maintenance at your site to determine any restrictions they may have imposed.

6.8.1. 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

6.8.1.1. 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.

6.8.1.1.1. 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.

6.8.1.1.1.1. 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.

6.8.1.1.1.2. 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.

6.8.1.1.1.3. 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.

6.8.1.1.1.4. 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.

6.8.1.1.1.5. 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).

6.8.1.1.1.6. 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.

6.8.1.1.1.7. 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.

6.8.1.2. 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.

6.9. CONVERT - Convert Disc Executive files

The CONVERT utility is a program which reads discs written by the Disc Executive (a simpler Marinchip operating system used in smaller single user configurations) and copies files from them to directories under the Network Operating System. CONVERT is invoked with the command:

```
CONVERT ⟨out directory⟩=⟨unit⟩/⟨selection⟩
```

where ⟨out directory⟩ is the name of the NOS directory file in which the converted files are to be placed. If the current assumed directory is to be the destination of the files, "." should be specified. ⟨unit⟩ is the number of the disc drive on which the Disc Executive disc has been previously mounted for arbitrary access. The ⟨selection⟩ specification may either name a single file name or designate a group of files by containing selection characters. 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 Disc Executive volume, any of the following specifications may be used:

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

Before a Disc Executive volume may be used with CONVERT, it must first be mounted for arbitrary access. The following example illustrates the use of CONVERT.

```
                                  Place Disc Executive disc
                                  in drive 2.
:MOUNT 2:*                        Mount for arbitrary access
:MAKEDIR DEXFILES                 Create directory for files
                                  to be converted.
:CONVERT DEXFILES=2/PROG.*        Convert all files with name
                                  PROG and any type.
:DISMOUNT 2:                      DISMOUNT Disc Exec disc.
                                  Remove disc from drive.
```

6.10. CREATE - Create file

The CREATE command allows creation of both normal and contiguous files from the user terminal. The format of the command is:

```
   CREATE ⟨file⟩[(⟨size⟩)],...
or CRE ⟨file⟩[(⟨size⟩)],...
```

where ⟨file⟩ is the name of the file to be created, optionally followed by a contiguous size specification in parentheses. If no size is specified, a normal file will be created. No space is assigned to a normal file upon creation, as the system will allocate space to the file as it is written into. If a contiguous file is to be created, the file name should be followed by the size required in bytes. The size is specified as a decimal number. For example, to allocate a contiguous file thirty million bytes long, one might use:

```
   CREATE LONGFILE(30000000)
```

If insufficient contiguous space exists on the volume to allocate the file, an error message will appear and no space will be allocated to the file.

Since most programs automatically create their output files, the CREATE command is used primarily to allocate contiguous files.

6.11. DELETE - Delete file

The DELETE command deletes a file from a directory. The format of the command is:

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

where ⟨file⟩ is the name of the file to be deleted. If the name deleted is the only name for a file, the space assigned to the file will be released to the free space pool on the volume.

6.12. DIRECT - List file directory

The DIRECT command lists file directory items. Either the item for a specific data file, or all the items in a directory file may be listed. The DIRECT command will follow nested directories to their ultimate nesting limits. The format of the command is:

```
   DIRECT [<options>]
   DIRECT [<options>]<volume/device>:
   DIRECT [<options>]<file>
or DIR    etc.
```

The first form lists all files in the user's assumed directory. The second form lists all files on the specified volume or device. The third form lists the directory item for the named file, if a data file. If a directory, it lists all files in the directory, continuing until all files which may be reached through that directory have been listed. When listing files, if a directory is encountered, the name of the directory will be listed, then DIRECT will indent the listing and list the contents of that directory.

The <options> field allows alternate presentation of the file directory. If an asterisk (*) precedes the specification, the time and date of creation and the last modification will be printed for each file listed. If a plus sign (+) precedes the specification, the directory will be printed on the printer configured as PRINT.DEV, rather than on the user's terminal. If both a plus sign and asterisk are used, they may be specified in either order.

The information printed for each file is in the format:

```
   name  size  owner  permissions  type
```

where "name" is the file name, "size" is the highest byte written in the file plus one (expressed in multiples of 1024 ("K") if larger than 32767), "owner" is the group and user number of the file's owner (enclosed in < > if the system is to assume the identity of the owner when the file is executed), "permissions" are the file permission bits (see below), and "type" is an optional message indicating various special file types.

The file permission bits are printed in the format:

```
   <global>-<group>-<owner>
```

where <global> indicates the operations permitted to users with a different group number than the file's owner, <group> specifies what members of the owner's group other than the owner may do, and

<owner> indicates the operations permitted for the file owner. The operations are indicated by the letters below:

R	Reading allowed
S	Search allowed (directories only)
W	Writing allowed
X	Execution allowed

6.13. 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 Network Operating System release disc, uses the disc handler within the Network Operating System. 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.

6.13.1. 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.

6.13.1.1. Mounting discs for access

Discs to be manipulated by DU must first be mounted for arbitrary access. This involves mounting the disc with a volume name of "*" which enables the unformatted mode of access to the volume used by DU. Normal system volumes cannot be accessed with DU. Refer to the description of the MOUNT system command and the MOUNT$ system call for more information.

6.13.1.2. 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 Network Operating System formatted discs, as the file directory addresses sectors by block number, rather than track and sector numbers.

6.13.1.2.1. 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.

6.13.1.2.2. 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.

6.13.1.2.3. 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.

6.13.1.2.4. 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.

6.13.1.2.5. DI - Dismount volume

```
DI ‹disc›
```

The Dismount command requests the operating system to dismount the volume mounted on the specified ‹disc›. If the request is rejected, the error code will be printed.

6.13.1.2.6. END - End disc utility

END

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

6.13.1.2.7. MO - Mount volume

MO ⟨disc⟩

The Mount command requests the operating system to mount the volume on the specified ⟨disc⟩ unit for arbitrary access. If the system rejects the mount request, the error code will be printed.

6.13.1.2.8. 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.

6.13.1.2.9. 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.

6.13.1.2.10. 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.

6.13.1.2.11. 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.

6.13.1.2.12. 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.

6.13.1.2.13. 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.

6.13.1.2.14. VT - Validate track

VT ⟨disc⟩,⟨track⟩

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

6.13.1.2.15. W - Write

W ⟨disc⟩,⟨track⟩,⟨sector⟩

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

6.13.1.2.16. 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.

6.14. 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 Network Operating System. 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.

6.14.1. 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 terminal.
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.

6.14.2. 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.

6.14.3. Temporary files

If the file being edited is larger than user memory, the editor will create the files "TEMP1$" and "TEMP2$" in the user's current directory to page the data. Both of these files must be large enough to hold the file being edited. If these files cannot be created, or free space on the volume holding the user's directory is exhausted, the message, "Buffer impasse.", will be issued, and additions to the file being edited will not be permitted. The user may direct the editor's temporary file usage to other files by creating ALIAS names of TEMP1$ and TEMP2$ in his working directory before calling the editor.

6.14.4. 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.

6.15. ERROR? - Edit message for error code

When an error occurs during the process of loading a program or executing a system command, the system will print an error message with a numeric code indicating the type of error. These codes are described in the section "System call error codes" earlier in this manual, but it is often inconvenient to have to refer to the manual when an error occurs. The command:

```
ERROR? ⟨code⟩
```

where ⟨code⟩ is the decimal error code will print a description of the error.

Various other system programs print the same error numbers when they encounter an error, and ERROR? may be used to determine the meaning of those errors as well.

6.16. 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. FDIAG is intended to be run only on contiguous files, as otherwise it would fill the entire volume with data, then error when all space on the volume was exhausted.

6.16.1. 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.

6.16.2. 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.

6.17. 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.

6.17.1. 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.

6.17.1.1. 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
```

6.17.1.2. 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 (#).

6.17.1.2.1. 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.

6.17.1.2.2. 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.

6.17.1.2.3. 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
```

6.17.1.2.4. 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.

6.17.1.2.5. 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.

6.17.1.2.6. 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.

6.17.1.3. 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.

6.17.1.4. 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.

6.17.1.5. 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.

6.17.2. 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
```

6.17.3. 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.

6.18. LOGIN - Log on to system

The LOGIN command is automatically activated by the system whenever a new terminal connects. LOGIN requests identification from the user, and only allows access to the system if the user name and password are correct. If desired, LOGIN will automatically assume the default directory for the user, and activate a standard program. LOGIN may be called explicitly by a user to sign off and sign on under a new user name.

6.18.1. Using LOGIN

When LOGIN is run (whether automatically by the system or explicitly called by the user), it will prompt the user with:

```
Enter user name:
```

The user should respond with his user name, which is a string of from one to fourteen characters. The case of these characters is insignificant. LOGIN will search the database of users known to the system, and if the user name is found, respond with:

```
Enter password:
```

The user must then enter the correct password for the user name. Passwords, like user names, are from one to fourteen characters in length, and are matched regardless of case. The password will not be echoed to the user terminal when it is entered. Since the user cannot see what is being typed, none of the local editing keys will work while the password is being entered. Pressing any of the local editing keys will result in the "Enter password:" prompt being retyped and all password characters typed up to that point being discarded.

If the password is correct, the user will be logged onto the system and a message will be displayed confirming the log on and giving the user's group and user number. If the password is incorrect, a message to this effect will be printed and the password prompt will be repeated. The entire login attempt may be restarted at any time by pressing Control C, which will get the user back the the "Enter user name:" prompt.

6.18.2. User database file maintenance

The following information describes the format of the user database file. This information is relevant only to those who maintain this file. General users of the system need not bother with these details.

The user database is read from the file:

```
1:UTIL/USERDATABASE
```

This file is created with reading and writing only by owner, under group and user zero. Such a file can be read and written only by privileged users and programs. LOGIN is a privileged program, and thus is able to access the database. System maintenance personnel may update the user database by signing on with a privileged user name.

The user database is a standard text file, consisting of one or more lines. Each line describes a user name. The format of these lines is:

```
⟨username⟩    ⟨password⟩    ⟨parameters⟩
```

The ⟨username⟩ and ⟨password⟩ fields are both fixed-format 14 characters fields, and must be in UPPER CASE. The ⟨username⟩ and ⟨password⟩ are simply the values to be typed in by the user to log in. The ⟨parameters⟩ field is a free format field starting in column 29 of the line, and has the format:

```
⟨group⟩,⟨user⟩,⟨privacy⟩[,⟨defdir⟩[,⟨initprog⟩]]
```

where ⟨group⟩ is the group number for the user name, ⟨user⟩ is the user number, and ⟨privacy⟩ is the privacy bit settings to be used on a CREATE$ JSYS when the ⟨privacy modes⟩ field in the JSYS packet is zero. ⟨group⟩, ⟨user⟩, and ⟨privacy⟩ are all numeric values, and follow the normal system convention that a leading zero denotes a hexadecimal value.

The optional ⟨defdir⟩ field is the name of the user's working directory file. If this field is specified, LOGIN will do an ASDIR$ function on the name specified in that field. If the optional ⟨initprog⟩ field is present, LOGIN will execute the command in that field upon completing the log in process, rather than returning the user to system command mode. Use of this feature permits users to be "locked in" to certain application programs. It is possible to specify an ⟨initprog⟩ without specifying a ⟨defdir⟩ by preceding the ⟨initprog⟩ field by two commas.

To illustrate the format of the user database, consider the following user file entries:

```
SYSMAINT      DUCK          0,0,02F
BUSTER        927KDT        112,278,7,UVOL:UFILES/BUSTER
ACCOUNTING    MONEY         201,1904,02F,1:ACC,EDIT ACCDB
```

The first line is an entry for a user with name "SYSMAINT". The password for this user is "DUCK", and this user runs on group 0, user 0, which is the privileged user number. The specification of 02F sets the default privacy for that user. (Refer to the discussion of privacy bits in the description of the CREATE$ system call for an explanation of this value.)

The second line in the file is a typical user entry. User "BUSTER" has a password of "927KDT", group number 112, user number 278, default privacy of 7, and working directory UVOL:UFILES/BUSTER.

The third line illustrates an "application" user. This user will automatically have the command:

```
EDIT ACCDB
```

executed when the log in process is complete.

6.19. MAKEDIR - Make directory file

The MAKEDIR command creates a directory file, or can be used to turn an empty data file into a directory file. The format of the command is:

```
MAKEDIR ⟨file⟩,...
```

where ⟨file⟩ is the name of the directory to be created.

When using MAKEDIR as a privileged user to create initial directories for new users, be sure to use OWNER to change the ownership of each newly created directory to the group and user numbers to be assigned to the new user. Failure to do this will result in the user being unable to assign his own directory, since it will have been created by the privileged user and carry the privileged user's group and user numbers.

6.20. MCOPY - Multiple File Copy Utility

The MCOPY utility is designed to copy multiple disc files from one NOS directory to another. The output files are made as much as possible like the input files: access bits are the same; the output is a contiguous file if the input is; ownership is the same (if the utility is executed by a privileged user). If one input file is an alias of another, the output will be aliased files rather than separate copies.

Here are some of the operations that MCOPY can perform:

- Copy all the files from one directory to another, including directories, the files in those directories, and so on.
- Copy all the files from one disc to another (just a special case of the above).
- Copy all the files with names that begin with A and end with .REL
- Copy all the data files but none of the directories that appear in a given directory.

6.20.1. Using MCOPY

The program is in the file BIN on the NOS release disc and is invoked simply by typing its name with the specifications for the files to be copied. The simplest (and usual) way of using it is as follows:

```
MCOPY ⟨output directory⟩=⟨input directory⟩/⟨mask⟩
```

The ⟨mask⟩ selects the filenames to be copied, according to the rules given below. In particular, if it is null, everything will be copied. Notice the difference between the two following examples, which illustrate one of the easiest ways of going astray:

```
MCOPY 2:=1:UFILES/JOE-SMUDLEY
MCOPY 2:=1:UFILES/JOE-SMUDLEY/
```

The first will copy file JOE-SMUDLEY from directory 1:UFILES to the root directory of unit 2. The second will copy all files from directory 1:UFILES/JOE-SMUDLEY to the root directory of unit 2.

The directory names given in a call on MCOPY can use the full generality of NOS file names, using the assumed directory or fully

qualified file names. For instance, output can be put in the assumed directory by leaving the ⟨output directory⟩ blank:

```
MCOPY =1:UFILES/JOE-SMUDLEY/
```

6.20.2. File selection masks

The file selection mask can name a specific file, as above, or use special characters to select a whole class of files. The special characters are similar but not identical to those used by the DIR and DEL directives in the Disc Executive system:

*	Matches zero or more characters, up to the next period or the end of the name.
?	Matches any one character except period.
.	Matches a period or the end of the name.

Tne special treatment of periods is based on the normal use of period to indicate a file type, such as .REL, .ASM, etc. If the rules seem tricky, some examples should make them clear:

.	Matches any name, provided it does not have two periods; e.g., A, A.REL, SYZYGY.LONG
*	Matches any name which does not contain a period; e.g., A, ABRACADABRA, Z234567890123
AB*.*	Matches any name beginning with AB; e.g., AB, ABACUS.SIM, ABRACADABRA
A?.*	Matches any two-letter name beginning with A, with any file type; e.g., AB, AB.REL, A5.BAS -- but not ABC or ABACUS.SIM
.?	Matches any name with a file type of at least one character; e.g., PROGRAM.BAS, XYZ.Q

If the ⟨input directory⟩ contains directories, the MCOPY program will copy them and all files that they contain, down to any level. Thus,

```
MCOPY 2:=1:
```

when executed by a privileged user will make an exact copy of disc 1. However, it is possible to select directories as well as data files by means of a mask: instead of using ⟨mask⟩ one uses ⟨directory mask⟩\⟨data mask⟩ as in

```
MCOPY 2:A=1:UFILES/A*\*.REL
```

which would copy only directories beginning with A and data files ending with .REL. The normal use of this feature is to suppress

copying of directories by giving an impossible ⟨directory mask⟩:

```
MCOPY 2:A=1:UFILES/JOE-SMUDLEY/!\*.BAS
```

6.20.3. Multiple specifications

A call on MCOPY can have several specifications separated by commas:

```
MCOPY 2:RELOC=1:A/*.REL,2:RELOC=1:B/*.REL
```

Usually the only difference between this call and two separate calls on MCOPY is that it avoids loading the program two times. However, there is one subtle difference: MCOPY remembers all the files that it has copied, and uses aliases wherever it is appropriate. In the example above, suppose that 1:A/X.REL is an alias of 1:B/Y.REL; then 2:RELOC/X.REL will be an alias of 2:RELOC/Y.REL. If MCOPY had been called two separate times, 2:RELOC/X.REL and 2:RELOC/Y.REL would have been separate copies of the same file.

The example shows two copies into the same directory, but there is no restriction on the directories or units used in multiple specifications.

6.20.4. Error messages

There are two types of error message from MCOPY.

Error. Use:
MCOPY ⟨out directory⟩=⟨in directory⟩/⟨mask⟩, . . .

This message appears when there is a syntax error in the call to MCOPY. This could result from multiple specifications separated by something other than commas, no specifications at all, or a badly formed ⟨input⟩/⟨mask⟩.

Unable to ⟨do something to⟩ file ⟨name⟩. Status = ⟨number⟩.

This message results from the failure of some operation necessary for the copying of a file. The ⟨number⟩ is the status which was returned by NOS. Typical errors are protection violations, non-existence of ⟨input directory⟩, and running out of room on the output disc. After such an error, MCOPY will continue trying to copy files.

6.21. OWNER - Change file ownership

The OWNER command allows a privileged user to change the ownership of an existing file. The format of the command is:

```
OWNER ⟨file⟩(⟨group⟩,⟨user⟩),...
```

where ⟨file⟩ is the name of the file to be changed, ⟨group⟩ is the new group number, and ⟨user⟩ is the new user number.

6.22. 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 Network Operating System, permitting it to interchange files with all other Marinchip software.

6.22.1. Calling the compiler

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

PASCAL(⟨source⟩,⟨listing⟩,⟨object⟩[,⟨temp1⟩,⟨temp2⟩])

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. The ⟨temp1⟩ and ⟨temp2⟩ specifications are optional, and may be used to direct the compiler temporary files to nonstandard file names. See the section "Temporary files" below for more information on these specifications.

6.22.2. Executing the program

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

6.22.3. Temporary files

The Pascal compiler uses two temporary files to hold intermediate output generated during compilation of a program. The compiler will normally use two files named TEMP1$ and TEMP2$, which it expects to find in the user's current working directory. If other files are to be used, they may be specified via the ⟨temp1⟩ and ⟨temp2⟩ optional specifications when the compiler is called.

6.22.4. For more information

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

6.23. PREP - Write initial file directory

The PREP command is used by privileged users to create empty file directories on blank volumes, or on volumes which are to be reused. The format of the command is:

```
PREP [*]⟨device⟩:⟨volume⟩[(⟨size⟩,⟨count⟩)],...
```

where ⟨device⟩ is the explicit storage unit device name (such as "1"), and ⟨volume⟩ is the volume name to be used on the volume being PREPed. If specified, ⟨size⟩ is the size of data blocks on the volume in bytes, and ⟨count⟩ is the number of such blocks present on the volume. If ⟨size⟩ and ⟨count⟩ are omitted, the default values for the storage unit will be used. Normally, PREP will perform a complete surface test of the volume, and remove any bad blocks from eligibility for allocation to user files. If a leading asterisk is specified before the ⟨device⟩ name, this test will be omitted, and all blocks will be assumed to be good.

PREP may only be performed by users running under group number 0 and user number 666, and may be done only on volumes mounted on storage units for arbitrary access ("*" as volume name).

6.24. SCRIPT - Execute commands from file

The SCRIPT command lets the user cause the system to execute commands from a file instead of requiring that each command be entered on the user terminal. The format of the command is:

SCRIPT ⟨file⟩

where ⟨file⟩ is the name of the text file containing the commands to be executed. If the file executes a program which normally reads its input from the terminal, that program will read from the ⟨file⟩ specified on the SCRIPT command. Thus, the user may prepare an entire file containing work for the system to do, start it up, and let the system execute the script without intervention from the user. Since the ⟨file⟩ specified may be generated by a program, this allows user programs to generate sequences of work to be done by the system.

A SCRIPT command may appear within a script. The maximum nesting permitted is specified when the system is generated. Note that the maximum applies to the sum of currently effective SCRIPT and user program generated SCRIPT$ requests. (The SCRIPT command actually works simply by submitting a SCRIPT$ request, so there is actually only one limit.)

6.25. 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 terminal.
- Making a hard copy of a listing stored on disc.

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

6.25.1. 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.

6.25.1.1. Examples of use

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

```
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
```

6.25.1.2. 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.

6.26. TIME - Set / Display system time

The TIME program may be used by any user to display the current date and time, and by privileged users to set the system time. To display the time, the command is simply:

```
TIME
```

To set the time, the program should be called as follows:

```
TIME YYYY/MM/DD HH:MM:SS
```

where YYYY is the current year, MM is the current month (1 to 12), and day is the current day (1 to 31). HH, MM, and SS are the hours, minutes, and seconds, respectively of the Greenwich Mean Time to which the clock is to be set. When the command is entered, the caller will be prompted with a message which informs him to respond when the time entered is correct. The time entered on the TIME command is normally a minute or so ahead of the current time. After the command is entered and the prompt is displayed, the caller should wait until the exact time is reached, then respond to the prompt, which will set the clock very accurately.

6.27. VSTAT - Print volume status

The VSTAT command prints the status of a storage unit and the volume which is mounted on it. Either the status of one unit or all units configured in the system can be printed. VSTAT is called as follows:

```
   VSTAT (unit):,...
or VSTAT (volume):,...
or VSTAT
```

If no specifications are given on the VSTAT command, the status of all configured units will be printed. If a file system volume is mounted on a unit, the status may be requested by volume name. In any case, the unit number can be specified.

VSTAT will print the unit number, followed by a status description which depends upon the unit status. If no volume is mounted, the unit will be said to be idle. If a volume is mounted for arbitrary access (asterisk as volume name), a message will be printed to that effect. If a normal file system volume is mounted, its volume name will be printed and the amount of free space and the largest contiguous block in bytes will be printed. The free space and largest block will be printed to the byte if less than 32768 and in units of 1024 bytes (1 "K") if larger.

6.28. 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

6.28.1. 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.

6.28.2. 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.

7. 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.

The system relocatable library consists of files stored in the directory:

```
RELOC
```

on the system volume. Hence, to include the relocatable file TRACE.REL in a program, the following LINK command would be used:

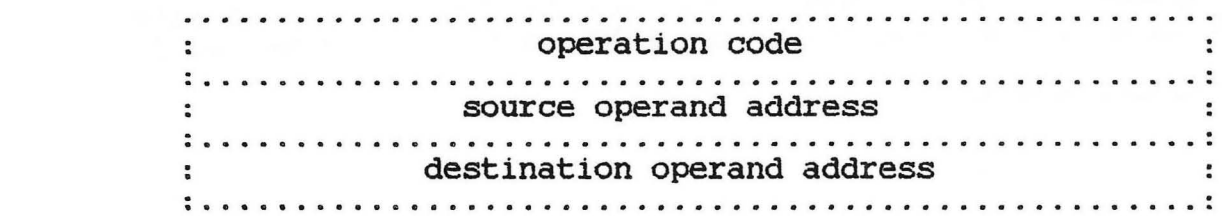
```
IN 1:RELOC/TRACE.REL
```

7.1. DFLOAT.REL - Double precision floating

Entry points: FPD$, FPDR$

The double precision floating point routines emulate IBM System/370 double precision floating point operations. Double precision numbers are 64 bits in length (8 bytes), with a sign, 7 bit hexadecimal exponent, and 56 bits of mantissa. Double precision arithmetic provides approximately 16 decimal digits of accuracy.

All communication with the double precision floating point subroutines is through a request packet with the following format:



The ⟨operation code⟩ field contains a code for the floating point operation to be performed. The codes are defined below, and the mnemonics used are defined in the file:

```
1:SOURCE/JSYS$
```

which also defines the system call codes.

Mnemonic	Code	Meaning
AE$	1	Add (dest=dest+source)
SE$	2	Subtract (dest=dest-source)
ME$	3	Multiply (dest=dest*source)
DE$	4	Divide (dest=dest/source)
CE$	5	Compare (source:dest)

The ⟨source operand address⟩ field specifies the address where the 8 byte source operand number starts, and the ⟨destination operand address⟩ field specifies the start address for the 8 byte destination operand number.

Two different calling sequences are available for the double precision floating point package. The first version uses a static packet address following the call, as follows:

```
BLWP    FPD$
DATA    ⟨packet address⟩
```

```
        <return>
```

This calling sequence has the advantage that no user workspace registers are required to be preloaded before the call, and no registers are changed by the call, but is unsuitable for reentrant calls with dynamically allocated packets, as the packet address would have to be stored into the program. For such "dynamic packet" applications, the call:

```
        LI      R0,<packet address>
        BLWP    FPDR$
        <return>
```

may be used. This call is identical to FPD$ except that the packet address is taken from user workspace register R0 instead of the word following the call instruction, and return is made immediately after the call instruction.

The arithmetic calls, AE$, SE$, ME$, and DE$, do not affect the condition bits unless an error occurs during their execution. A divide fault or exponent overflow will set the destination operand to the largest possible number (with the sign of the result) and will set the overflow status bit. An exponent underflow will clear the destination to zero, and will set the equal and overflow status bits.

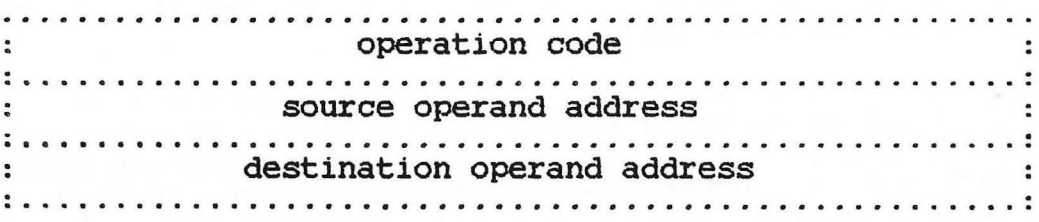
The comparison call, CE$, will set the condition bits based on the relation between the source and destination operands. If the source and destination are equal, the equal status bit will be set. If the source is greater, the arithmetic and logical greater than bits will be set. If the source is less, no bits will be set. Note that the comparison done by CE$ is always a signed comparison, but sets the arithmetic and logical status bits the same. This is done so that the more flexible logical status test instructions (JHE, JL, etc.) may be used to test the result of a floating point comparison, as well as the arithmetic test instructions (JGT, JLT).

7.2. FLOAT.REL - Single precision floating

Entry points: FPS$, FPSR$

The single precision floating point routines emulate IBM System/370 single precision floating point operations. Single precision numbers are 32 bits in length (4 bytes), with a sign, 7 bit hexadecimal exponent, and 24 bits of mantissa. Single precision arithmetic provides between 6 and 7 decimal digits of accuracy.

All communication with the single precision floating point subroutines is through a request packet with the following format:



The ⟨operation code⟩ field contains a code for the floating point operation to be performed. The codes are defined below, and the mnemonics used are defined in the file:

```
1:SOURCE/JSYS$
```

which also defines the system call codes.

Mnemonic	Code	Meaning
AE$	1	Add (dest=dest+source)
SE$	2	Subtract (dest=dest-source)
ME$	3	Multiply (dest=dest*source)
DE$	4	Divide (dest=dest/source)
CE$	5	Compare (source:dest)

The ⟨source operand address⟩ field specifies the address where the 4 byte source operand number starts, and the ⟨destination operand address⟩ field specifies the start address for the 4 byte destination operand number.

Two different calling sequences are available for the single precision floating point package. The first version uses a static packet pointer following the call, as follows:

```
        BLWP    FPS$
        DATA    ⟨packet address⟩
```

```
        <return>
```

This calling sequence has the advantage that no user workspace registers are required to be preloaded before the call, and no registers are changed by the call, but is unsuitable for reentrant calls with dynamically allocated packets, as the packet address would have to be stored into the program. For such "dynamic packet" applications, the call:

```
        LI      R0,<packet address>
        BLWP    FPSR$
        <return>
```

may be used. This call is identical to FPS$ except that the packet address is taken from user workspace register R0 instead of the word following the call instruction, and return is made immediately after the call instruction.

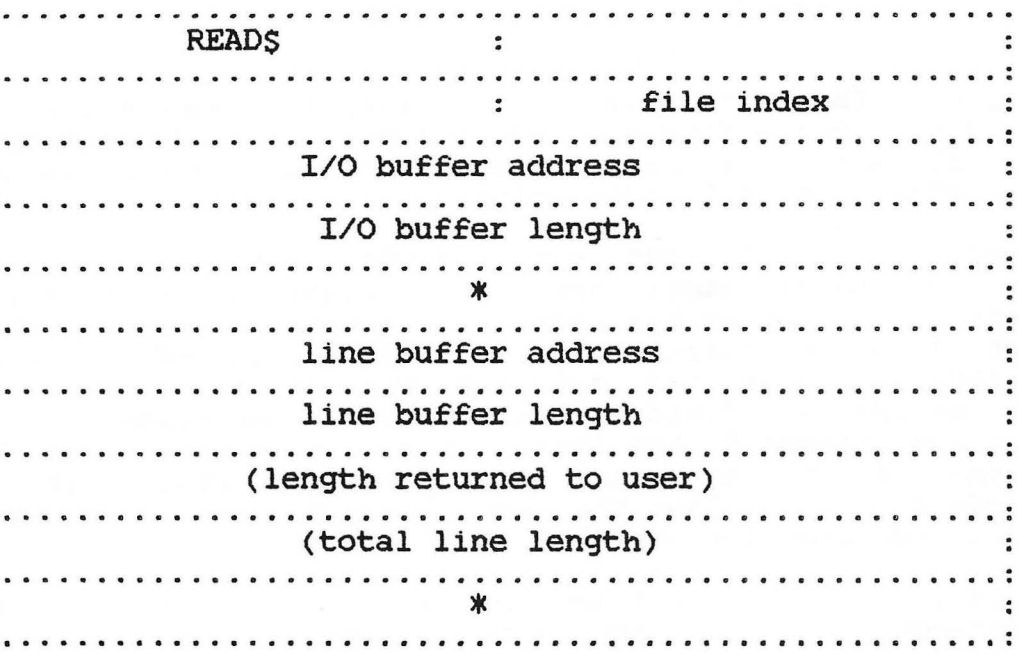
The arithmetic calls, AE$, SE$, ME$, and DE$, set the condition bits based on the arithmetic and logical magnitude of the result of the operation, based on the normal definitions of these bits (note that this is different from the action of the double precision package, which does not set the condition bits for arithmetic calls). A divide fault or exponent overflow will set the destination operand to the largest possible number (with the sign of the result) and will set the overflow status bit. An exponent underflow will clear the destination to zero, and will set the equal and overflow status bits.

The comparison call, CE$, will set the condition bits based on the relation between the source and destination operands. If the source and destination are equal, the equal status bit will be set. If the source is greater, the arithmetic and logical greater than bits will be set. If the source is less, no bits will be set. Note that the comparison done by CE$ is always a signed comparison, but sets the arithmetic and logical status bits the same. This is done so that the more flexible logical status test instructions (JHE, JL, etc.) may be used to test the result of a floating point comparison, as well as the arithmetic test instructions (JGT, JLT).

7.3. 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.

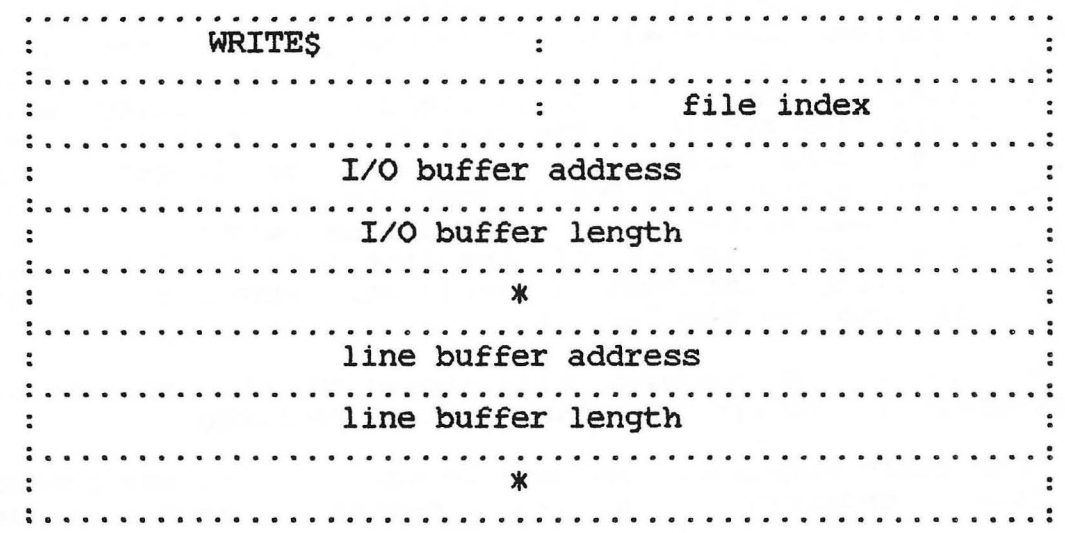
TEXTIN is completely reentrant, and may be used to read any number of text files concurrently (using one packet for each file, of course).

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.

7.4. 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.

7.5. 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.